Table of Contents

An Introduction to Elm

Elm is a functional language that compiles to JavaScript. It competes with projects like React as a tool for creating websites and web apps. Elm has a very strong emphasis on simplicity, ease-of-use, and quality tooling.

This guide will:

- Teach you the fundamentals of programming in Elm.
- Show you how to make interactive apps with *The Elm Architecture*.
- Emphasize the principles and patterns that generalize to programming in any language.

By the end I hope you will not only be able to create great web apps in Elm, but also understand the core ideas and patterns that make Elm nice to use.

If you are on the fence, I can safely guarantee that if you give Elm a shot and actually make a project in it, you will end up writing better JavaScript and React code. The ideas transfer pretty easily!

A Quick Sample

Of course *I* think Elm is good, so look for yourself.

Here is a simple counter. If you look at the code, it just lets you increment and decrement the counter:

```
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Html.beginnerProgram { model = 0, view = view, update = update }

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Notice that the `update` and `view` are entirely decoupled. You describe your HTML in a declarative way and Elm takes care of messing with the DOM.

Why a *functional* language?

Forget what you have heard about functional programming. Fancy words, weird ideas, bad tooling. Barf. Elm is about:

- No runtime errors in practice. No `null`. No `undefined` is not a function.
- Friendly error messages that help you add features more quickly.
- Well-architected code that *stays* well-architected as your app grows.
- Automatically enforced semantic versioning for all Elm packages.

No combination of JS libraries can ever give you this, yet it is all free and easy in Elm. Now these nice things are *only* possible because Elm builds upon 40+ years of work on typed functional languages. So Elm is a functional language because the practical benefits are worth the couple hours you'll spend reading this guide.

I have put a huge emphasis on making Elm easy to learn and use, so all I ask is that you give Elm a shot and see what you think. I hope you will be pleasantly surprised!

> **Note:** If you do not want to install yet, you can follow along in this guide with the online editor and the online REPL.

Install

- Mac — installer
- Windows — installer
- Anywhere — npm installer or build from source

After installing through any of those routes, you will have the following command line tools:

- `elm-repl` — play with Elm expressions
- `elm-reactor` — get a project going quickly
- `elm-make` — compile Elm code directly
- `elm-package` — download packages

We will go over how they all work in more detail right after we get your editor set up!

> **Troubleshooting:** The fastest way to learn *anything* is to talk with other people in the Elm community. We are friendly and happy to help! So if you get stuck during installation or encounter something weird, visit the Elm Slack and ask about it. In fact, if you run into something confusing at any point while learning or using Elm, come ask us about it. You can save yourself hours. Just do it!

Configure Your Editor

Using Elm is way nicer when you have a code editor to help you out. There are Elm plugins for at least the following editors:

- Atom
- Brackets
- Emacs
- IntelliJ
- Light Table
- Sublime Text
- Vim
- VS Code

If you do not have an editor at all, Sublime Text is a great one to get started with!

You may also want to try out elm-format which makes your code pretty!

The Command Line Tools

So we installed Elm, and it gave us `elm-repl` , `elm-reactor` , `elm-make` , and `elm-package` . But what do they all do exactly?

elm-repl

`elm-repl` lets you play with simple Elm expressions.

```
$ elm-repl
---- elm-repl 0.18.0 -----------------------------------------------------------
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
> 1 / 2
0.5 : Float
> List.length [1,2,3,4]
4 : Int
> String.reverse "stressed"
"desserts" : String
> :exit
$
```

We will be using `elm-repl` in the upcoming “Core Language” section, and you can read more about how it works here.

> **Note:** `elm-repl` works by compiling code to JavaScript, so make sure you have Node.js installed. We use that to evaluate code.

elm-reactor

`elm-reactor` helps you build Elm projects without messing with the command-line too much. You just run it at the root of your project, like this:

```
git clone https://github.com/evancz/elm-architecture-tutorial.git
cd elm-architecture-tutorial
elm-reactor
```

This starts a server at `http://localhost:8000` . You can navigate to any Elm file and see what it looks like. Try to check out `examples/01-button.elm` .

Notable flags:

- `--port` lets you pick something besides port 8000. So you can say `elm-reactor --port=8123` to get things to run at `http://localhost:8123` .

- `--address` lets you replace `localhost` with some other address. For example, you may want to use `elm-reactor --address=0.0.0.0` if you want to try out an Elm program on a mobile device through your local network.

elm-make

`elm-make` builds Elm projects. It can compile Elm code to HTML or JavaScript. It is the most general way to compile Elm code, so if your project becomes too advanced for `elm-reactor`, you will want to start using `elm-make` directly.

Say you want to compile `Main.elm` to an HTML file named `main.html`. You would run this command:

```
elm-make Main.elm --output=main.html
```

Notable flags:

- `--warn` prints warnings to improve code quality

elm-package

`elm-package` downloads and publishes packages from our package catalog. As community members solve problems in a nice way, they share their code in the package catalog for anyone to use!

Say you want to use `elm-lang/http` and `NoRedInk/elm-decode-pipeline` to make HTTP requests to a server and turn the resulting JSON into Elm values. You would say:

```
elm-package install elm-lang/http
elm-package install NoRedInk/elm-decode-pipeline
```

This will add the dependencies to your `elm-package.json` file that describes your project. (Or create it if you do not have one yet!) More information about all this here!

Notable commands:

- `install`: install the dependencies in `elm-package.json`
- `publish`: publish your library to the Elm Package Catalog
- `bump`: bump version numbers based on API changes
- `diff`: get the difference between two APIs

Core Language

This section will walk you through Elm's simple core language.

This works best when you follow along, so after installing, start up `elm-repl` in the terminal. (Or use the online REPL.) Either way, you should see something like this:

```
---- elm repl 0.18.0 -----------------------------------------------------------
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
>
```

The REPL prints out the type of every result, but **we will leave the type annotations off in this tutorial** for the sake of introducing concepts gradually.

We will cover values, functions, lists, tuples, and records. These building blocks all correspond pretty closely with structures in languages like JavaScript, Python, and Java.

Values

Let's get started with some strings:

```
> "hello"
"hello"

> "hello" ++ "world"
"helloworld"

> "hello" ++ " world"
"hello world"
```

Elm uses the `(++)` operator to put strings together. Notice that both strings are preserved exactly as is when they are put together so when we combine `"hello"` and `"world"` the result has no spaces.

Math looks normal too:

```
> 2 + 3 * 4
14

> (2 + 3) * 4
20
```

Unlike JavaScript, Elm makes a distinction between integers and floating point numbers. Just like Python 3, there is both floating point division `(/)` and integer division `(//)`.

```
> 9 / 2
4.5

> 9 // 2
4
```

Functions

Let's start by writing a function `isNegative` that takes in some number and checks if it is less than zero. The result will be `True` or `False`.

```
> isNegative n = n < 0
<function>

> isNegative 4
False

> isNegative -7
True

> isNegative (-3 * -4)
False
```

Notice that function application looks different than in languages like JavaScript and Python and Java. Instead of wrapping all arguments in parentheses and separating them with commas, we use spaces to apply the function. So `(add(3,4))` becomes `(add 3 4)` which ends up avoiding a bunch of parens and commas as things get bigger. Ultimately, this looks much cleaner once you get used to it! The elm-html package is a good example of how this keeps things feeling light.

If Expressions

When you want to have conditional behavior in Elm, you use an if-expression.

```
> if True then "hello" else "world"
"hello"

> if False then "hello" else "world"
"world"
```

The keywords `if` `then` `else` are used to separate the conditional and the two branches so we do not need any parentheses or curly braces.

Elm does not have a notion of “truthiness” so numbers and strings and lists cannot be used as boolean values. If we try it out, Elm will tell us that we need to work with a real boolean value.

Now let's make a function that tells us if a number is over 9000.

```
> over9000 powerLevel = \
|   if powerLevel > 9000 then "It's over 9000!!!" else "meh"
<function>

> over9000 42
"meh"

> over9000 100000
"It's over 9000!!!"
```

Using a backslash in the REPL lets us split things on to multiple lines. We use this in the definition of `over9000` above. Furthermore, it is best practice to always bring the body of a function down a line. It makes things a lot more uniform and easy to read, so you want to do this with all the functions and values you define in normal code.

> **Note:** Make sure that you add a whitespace before the second line of the function. Elm has a "syntactically significant whitespace" meaning that indentation is a part of its syntax.

Lists

Lists are one of the most common data structures in Elm. They hold a sequence of related things, similar to arrays in JavaScript.

Lists can hold many values. Those values must all have the same type. Here are a few examples that use functions from [the `List` library]:

```
> names = [ "Alice", "Bob", "Chuck" ]
["Alice","Bob","Chuck"]

> List.isEmpty names
False

> List.length names
3

> List.reverse names
["Chuck","Bob","Alice"]

> numbers = [1,4,3,2]
[1,4,3,2]

> List.sort numbers
[1,2,3,4]

> double n = n * 2
<function>

> List.map double numbers
[2,8,6,4]
```

Again, all elements of the list must have the same type.

Tuples

Tuples are another useful data structure. A tuple can hold a fixed number of values, and each value can have any type. A common use is if you need to return more than one value from a function. The following function gets a name and gives a message for the user:

```
> import String

> goodName name = \
|   if String.length name <= 20 then \
|     (True, "name accepted!") \
|   else \
|     (False, "name was too long; please limit it to 20 characters")

> goodName "Tom"
(True, "name accepted!")
```

This can be quite handy, but when things start becoming more complicated, it is often best to use records instead of tuples.

Records

A record is a set of key-value pairs, similar to objects in JavaScript or Python. You will find that they are extremely common and useful in Elm! Let's see some basic examples.

```
> point = { x = 3, y = 4 }
{ x = 3, y = 4 }

> point.x
3

> bill = { name = "Gates", age = 57 }
{ age = 57, name = "Gates" }

> bill.name
"Gates"
```

So we can create records using curly braces and access fields using a dot. Elm also has a version of record access that works like a function. By starting the variable with a dot, you are saying *please access the field with the following name*. This means that `.name` is a function that gets the `name` field of the record.

```
> .name bill
"Gates"

> List.map .name [bill,bill,bill]
["Gates","Gates","Gates"]
```

When it comes to making functions with records, you can do some pattern matching to make things a bit lighter.

```
> under70 {age} = age < 70
<function>

> under70 bill
True

> under70 { species = "Triceratops", age = 68000000 }
False
```

So we can pass any record in as long as it has an `age` field that holds a number.

It is often useful to update the values in a record.

```
> { bill | name = "Nye" }
{ age = 57, name = "Nye" }

> { bill | age = 22 }
{ age = 22, name = "Gates" }
```

It is important to notice that we do not make *destructive* updates. When we update some fields of `bill` we actually create a new record rather than overwriting the existing one. Elm makes this efficient by sharing as much content as possible. If you update one of ten fields, the new record will share the nine unchanged values.

Comparing Records and Objects

Records in Elm are *similar* to objects in JavaScript, but there are some crucial differences. The major differences are that with records:

- You cannot ask for a field that does not exist.
- No field will ever be undefined or null.
- You cannot create recursive records with a `this` or `self` keyword.

Elm encourages a strict separation of data and logic, and the ability to say `this` is primarily used to break this separation. This is a systemic problem in Object Oriented languages that Elm is purposely avoiding.

Records also support structural typing which means records in Elm can be used in any situation as long as the necessary fields exist. This gives us flexibility without compromising reliability.

The Elm Architecture

The Elm Architecture is a simple pattern for architecting webapps. It is great for modularity, code reuse, and testing. Ultimately, it makes it easy to create complex web apps that stay healthy as you refactor and add features.

This architecture seems to emerge naturally in Elm. We first observed it in the games the Elm community was making. Then in web apps like TodoMVC and dreamwriter too. Now we see it running in production at companies like NoRedInk and Pivotal. The architecture seems to be a consequence of the design of Elm itself, so it will happen to you whether you know about it or not. This has proven to be really nice for onboarding new developers. Their code just turns out well-architected. It is kind of spooky.

So The Elm Architecture is *easy* in Elm, but it is useful in any front-end project. In fact, projects like Redux have been inspired by The Elm Architecture, so you may have already seen derivatives of this pattern. Point is, even if you ultimately cannot use Elm at work yet, you will get a lot out of using Elm and internalizing this pattern.

The Basic Pattern

The logic of every Elm program will break up into three cleanly separated parts:

- **Model** — the state of your application
- **Update** — a way to update your state
- **View** — a way to view your state as HTML

This pattern is so reliable that I always start with the following skeleton and fill in details for my particular case.

```
import Html exposing (..)


-- MODEL

type alias Model = { ... }


-- UPDATE

type Msg = Reset | ...

update : Msg -> Model -> Model
update msg model =
  case msg of
    Reset -> ...
    ...


-- VIEW

view : Model -> Html Msg
view model =
  ...
```

That is really the essence of The Elm Architecture! We will proceed by filling in this skeleton with increasingly interesting logic.

The Elm Architecture + User Input

Your web app is going to need to deal with user input. This section will get you familiar with The Elm Architecture in the context of things like:

- Buttons
- Text Fields
- Check Boxes
- Radio Buttons
- etc.

We will go through a few examples that build knowledge gradually, so go in order!

Follow Along

In the last section we used `elm-repl` to get comfortable with Elm expressions. In this section, we are switching to creating Elm files of our own. You can do that in the online editor, or if you have Elm installed, you can follow these simple instructions to get everything working on your computer!

Buttons

Clone the code or follow along in the online editor.

Our first example is a simple counter that can be incremented or decremented. I find that it can be helpful to see the entire program in one place, so here it is! We will break it down afterwards.

```
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)


main =
  Html.beginnerProgram { model = model, view = view, update = update }


-- MODEL

type alias Model = Int

model : Model
model =
  0


-- UPDATE

type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1


-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

That's everything!

> **Note:** This section has `type` and `type alias` declarations. You can read all about these in the upcoming section on types. You do not *need* to deeply understand that stuff now, but you are free to jump ahead if it helps.

When writing this program from scratch, I always start by taking a guess at the model. To make a counter, we at least need to keep track of a number that is going up and down. So let's just start with that!

```
type alias Model = Int
```

Now that we have a model, we need to define how it changes over time. I always start my `UPDATE` section by defining a set of messages that we will get from the UI:

```
type Msg = Increment | Decrement
```

I definitely know the user will be able to increment and decrement the counter. The `Msg` type describes these capabilities as *data*. Important! From there, the `update` function just describes what to do when you receive one of these messages.

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

If you get an `Increment` message, you increment the model. If you get a `Decrement` message, you decrement the model. Pretty straight-forward stuff.

Okay, so that's all good, but how do we actually make some HTML and show it on screen? Elm has a library called `elm-lang/html` that gives you full access to HTML5 as normal Elm functions:

```
view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

You can see more examples of basic HTML here.

One thing to notice is that our `view` function is producing a `Html Msg` value. This means that it is a chunk of HTML that can produce `Msg` values. And when you look at the definition, you see the `onClick` attributes are set to give out `Increment` and `Decrement` values. These will get fed directly into our `update` function, driving our whole app forward.

Another thing to notice is that `div` and `button` are just normal Elm functions. These functions take (1) a list of attributes and (2) a list of child nodes. It is just HTML with slightly different syntax. Instead of having `<` and `>` everywhere, we have `[` and `]`. We have found that folks who can read HTML have a pretty easy time learning to read this variation. Okay, but why not have it be *exactly* like HTML? **Since we are using normal Elm functions, we have the full power of the Elm programming language to help us build our views!** We can refactor repetitive code out into functions. We can put helpers in modules and import them just like any other code. We can use the same testing frameworks and libraries as any other Elm code. Everything that is nice about programming in Elm is 100% available to help you with your view. No need for a hacked together templating language!

There is also something a bit deeper going on here. **The view code is entirely declarative**. We take in a `Model` and produce some `Html`. That is it. There is no need to mutate the DOM manually, Elm takes care of that behind the scenes. This gives Elm much more freedom to make clever optimizations and ends up making rendering *faster* overall. So you write less code and the code runs faster. The best kind of abstraction!

This pattern is the essence of The Elm Architecture. Every example we see from now on will be a slight variation on this basic pattern: `Model`, `update`, `view`.

> **Exercise:** One cool thing about The Elm Architecture is that it is super easy to extend as our product requirements change. Say your product manager has come up with this amazing "reset" feature. A new button that will reset the counter to zero.
>
> To add the feature you come back to the `Msg` type and add another possibility: `Reset`. You then move on to the `update` function and describe what happens when you get that message. Finally you add a button in your view.
>
> See if you can implement the "reset" feature!

Text Fields

Clone the code or follow along in the online editor.

We are about to create a simple app that reverses the contents of a text field. This example also introduces some new stuff that will help us out in our next example.

Again this is a pretty short program, so I have included the whole thing here. Skim through to get an idea of how everything fits together. Right after that we will go into much more detail!

```
import Html exposing (Html, Attribute, div, input, text)
import Html.Attributes exposing (..)
import Html.Events exposing (onInput)


main =
  Html.beginnerProgram { model = model, view = view, update = update }


-- MODEL

type alias Model =
  { content : String
  }

model : Model
model =
  { content = "" }


-- UPDATE

type Msg
  = Change String

update : Msg -> Model -> Model
update msg model =
  case msg of
    Change newContent ->
      { model | content = newContent }


-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ input [ placeholder "Text to reverse", onInput Change ] []
    , div [] [ text (String.reverse model.content) ]
    ]
```

This code is a slight variant of the counter from the previous section. You set up a model. You define some messages. You say how to `update`. You make your `view`. The difference is just in how we filled this skeleton in. Let's walk through that!

As always, you start by guessing at what your `Model` should be. In our case, we know we are going to have to keep track of whatever the user has typed into the text field. We need that information so we know how to render the reversed text.

```
type alias Model =
  { content : String
  }
```

This time I chose to represent the model as a record. (You can read more about records here and here.) For now, the record stores the user input in the `content` field.

> **Note:** You may be wondering, why bother having a record if it only holds one entry? Couldn't you just use the string directly? Yes, of course! But starting with a record makes it easy to add more fields as our app gets more complicated. When the time comes where we want *two* text inputs, we will have to do much less fiddling around.

Okay, so we have our model. Now in this app there is only one kind of message really. The user can change the contents of the text field.

```
type Msg
  = Change String
```

This means our update function just has to handle this one case:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Change newContent ->
      { model | content = newContent }
```

When we receive new content, we use the record update syntax to update the contents of `content`.

Finally we need to say how to view our application:

```
view : Model -> Html Msg
view model =
  div []
    [ input [ placeholder "Text to reverse", onInput Change ] []
    , div [] [ text (String.reverse model.content) ]
    ]
```

We create a `<div>` with two children.

The interesting child is the `<input>` node. In addition to the `placeholder` attribute, it uses `onInput` to declare what messages should be sent when the user types into this input.

This `onInput` function is kind of interesting. It takes one argument, in this case the `Change` function which was created when we declared the `Msg` type:

```
Change : String -> Msg
```

This function is used to tag whatever is currently in the text field. So let's say the text field currently holds `yol` and the user types `o`. This triggers an `input` event, so we will get the message `Change "yolo"` in our `update` function.

So now we have a simple text field that can reverse user input. Neat! Now on to putting a bunch of text fields together into a more traditional form.

Forms

Clone the code or follow along in the online editor.

Here we will make a rudimentary form. It has a field for your name, a field for your password, and a field to verify that password. We will also do some very simple validation (do the two passwords match?) just because it is simple to add.

The code is a bit longer in this case, but I still think it is valuable to look through it before you get into the description of what is going on.

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (onInput)


main =
  Html.beginnerProgram { model = model, view = view, update = update }


-- MODEL

type alias Model =
  { name : String
  , password : String
  , passwordAgain : String
  }


model : Model
model =
  Model "" "" ""


-- UPDATE

type Msg
    = Name String
    | Password String
    | PasswordAgain String


update : Msg -> Model -> Model
update msg model =
```

```
  case msg of
    Name name ->
      { model | name = name }

    Password password ->
      { model | password = password }

    PasswordAgain password ->
      { model | passwordAgain = password }


-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ input [ type_ "text", placeholder "Name", onInput Name ] []
    , input [ type_ "password", placeholder "Password", onInput Password ] []
    , input [ type_ "password", placeholder "Re-enter Password", onInput PasswordAgain
 ] []
    , viewValidation model
    ]

viewValidation : Model -> Html msg
viewValidation model =
  let
    (color, message) =
      if model.password == model.passwordAgain then
        ("green", "OK")
      else
        ("red", "Passwords do not match!")
  in
    div [ style [("color", color)] ] [ text message ]
```

This is pretty much exactly how our text field example looked, just with more fields. Let's walk through how it came to be!

As always, you start out by guessing at the `Model`. We know there are going to be three text fields, so let's just go with that:

```
type alias Model =
  { name : String
  , password : String
  , passwordAgain : String
  }
```

Great, seems reasonable. We expect that each of these fields can be changed separately, so our messages should account for each of those scenarios.

```
type Msg
    = Name String
    | Password String
    | PasswordAgain String
```

This means our `update` is pretty mechanical. Just update the relevant field:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Name name ->
      { model | name = name }

    Password password ->
      { model | password = password }

    PasswordAgain password ->
      { model | passwordAgain = password }
```

We get a little bit fancier than normal in our `view` though.

```
view : Model -> Html Msg
view model =
  div []
    [ input [ type_ "text", placeholder "Name", onInput Name ] []
    , input [ type_ "password", placeholder "Password", onInput Password ] []
    , input [ type_ "password", placeholder "Re-enter Password", onInput PasswordAgain
 ] []
    , viewValidation model
    ]
```

It starts out normal: We create a `<div>` and put a couple `<input>` nodes in it. Each one has an `onInput` attribute that will tag any changes appropriately for our `update` function. (This is all building off of the text field example in the previous section.)

But for the last child we do not directly use an HTML function. Instead we call the `viewValidation` function, passing in the current model.

```
viewValidation : Model -> Html msg
viewValidation model =
  let
    (color, message) =
      if model.password == model.passwordAgain then
        ("green", "OK")
      else
        ("red", "Passwords do not match!")
  in
    div [ style [("color", color)] ] [ text message ]
```

This function first compares the two passwords. If they match, you want green text and a positive message. If they do not match, you want red text and a helpful message. With that info, we produce a `<div>` filled with a colorful message explaining the situation.

This starts to show the benefits of having our HTML library be normal Elm code. It would have looked really weird to jam all that code into our `view`. In Elm, you just refactor like you would with any other code!

On these same lines, you may notice that the `<input>` nodes all are created with pretty similar code. Say we made each input fancier: there is an outer `<div>` that holds a `<span>` and an `<input>` with certain classes. It would make total sense to break that pattern out into a `viewInput` function so you never have to repeat yourself. This also means you change it in one place and everyone gets the updated HTML.

> **Exercises:** One cool thing about breaking `viewValidation` out is that it is pretty easy to augment. If you are messing with the code as you read through this (as you should be!) you should try to:
>
> - Check that the password is longer than 8 characters.
> - Make sure the password contains upper case, lower case, and numeric characters.
> - Add an additional field for `age` and check that it is a number.
> - Add a "Submit" button. Only show errors *after* it has been pressed.
>
> Be sure to use the helpers in the `String` library if you try any of these! Also, we need to learn more before we start talking to servers, so before you try that here, keep reading until HTTP is introduced. It will be significantly easier with proper guidance!

More About User Input

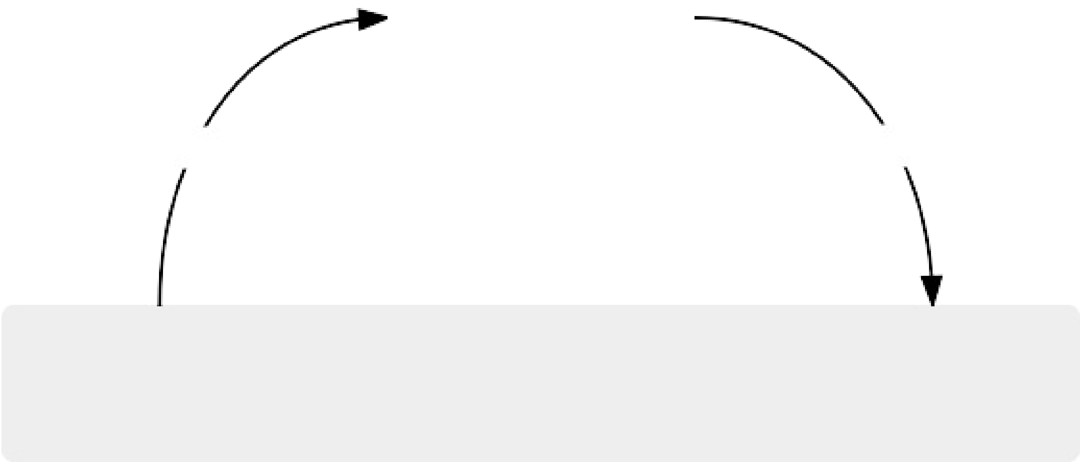
We only covered buttons and text fields, but there are other crazier inputs that you will need eventually.

So if you want to see examples of radio buttons and check boxes, visit the Elm examples page which has a bunch of small examples you can mess around with. It is all variations on stuff we have learned in this tutorial already, so playing with these examples is a great way to practice and become more comfortable with what you have learned so far. Maybe try incorporating check boxes into the form example?

That said, I want to keep up the momentum of this tutorial and keep introducing new concepts, so next we will be looking at how to work with things like HTTP and web sockets!

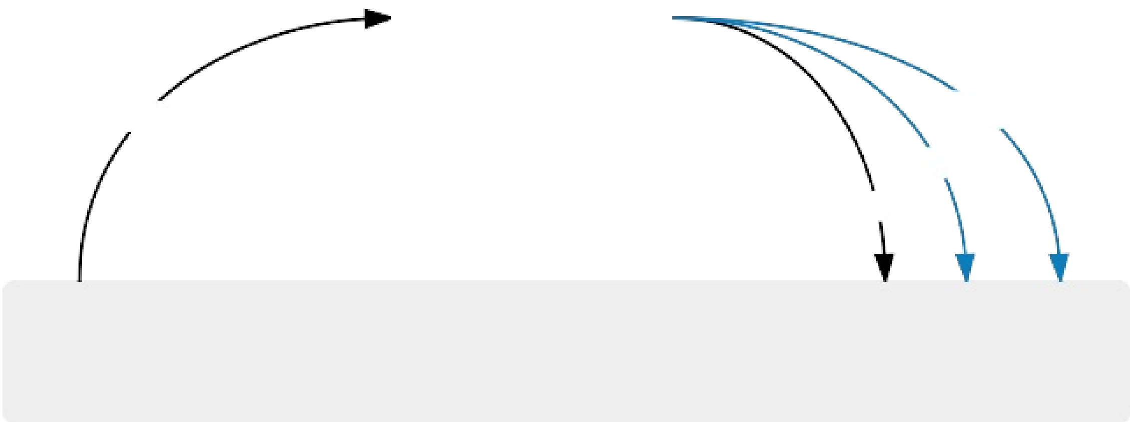
The Elm Architecture + Effects

The last section showed how to handle all sorts of user input. You can think of those programs like this:



From our perspective, we just receive messages and produce new `Html` to get rendered on screen. The “Elm Runtime” is sitting there behind the scenes. When it gets `Html` it figures out how to render it on screen really fast. When a user clicks on something, it figures out how to pipe that into our program as a `Msg`. So the Elm Runtime is in charge of *doing* stuff. We just transform data.

This section builds on that pattern, giving you the ability to make HTTP requests or subscribe to messages from web sockets. Think of it like this:



Instead of just producing `Html`, we will now be producing commands and subscriptions:

- **Commands** — A `Cmd` lets you *do* stuff: generate a random number, send an HTTP request, etc.
- **Subscriptions** — A `Sub` lets you register that you are interested in something: tell me about location changes, listen for web socket messages, etc.

If you squint, commands and subscriptions are pretty similar to `Html` values. With `Html`, we never touch the DOM by hand. Instead we represent the desired HTML as *data* and let the Elm Runtime do some clever stuff to make it render really fast. It is the same with commands and subscriptions. We create data that *describes* what we want to do, and the Elm Runtime does the dirty work.

Don't worry if it seems a bit confusing for now, the examples will help! So first let's look at how to fit these concepts into the code we have seen before.

Extending the Architecture Skeleton

So far our architecture skeleton has focused on creating `Model` types and `update` and `view` functions. To handle commands and subscriptions, we need to extend the basic architecture skeleton a little bit:

```
-- MODEL

type alias Model =
  { ...
  }


-- UPDATE

type Msg = Submit | ...

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  ...


-- VIEW

view : Model -> Html Msg
view model =
  ...


-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  ...


-- INIT

init : (Model, Cmd Msg)
init =
  ...
```

The first three sections are almost exactly the same, but there are a few new things overall:

1. The `update` function now returns more than just a new model. It returns a new model and some commands you want to run. These commands are all going to produce `Msg` values that will get fed right back into our `update` function.

2. There is a `subscriptions` function. This function lets you declare any event sources you need to subscribe to given the current model. Just like with `Html Msg` and `Cmd Msg`, these subscriptions will produce `Msg` values that get fed right back into our `update` function.

3. So far `init` has just been the initial model. Now it produces both a model and some commands, just like the new `update`. This lets us provide a starting value *and* kick off any HTTP requests or whatever that are needed for initialization.

Now it is totally okay if this does not really make sense yet! That only really happens when you start seeing it in action, so let’s hop right into the examples!

> **Aside:** One crucial detail here is that commands and subscriptions are *data*. When you create a command, you do not actually *do* it. Same with commands in real life. Let’s try it. Eat an entire watermelon in one bite! Did you do it? No! You kept reading before you even *thought* about buying a tiny watermelon.
>
> Point is, commands and subscriptions are data. You hand them to Elm to actually run them, giving Elm a chance to log all of this information. In the end, effects-as-data means Elm can:
>
> - Have a general purpose time-travel debugger.
> - Keep the "same input, same output" guarantee for all Elm functions.
> - Avoid setup/teardown phases when testing `update` logic.
> - Cache and batch effects, minimizing HTTP connections or other resources.
>
> So without going too crazy on details, pretty much all the nice guarantees and tools you have in Elm come from the choice to treat effects as data! I think this will make more sense as you get deeper into Elm.

Random

Clone the code or follow along in the online editor.

We are about to make an app that "rolls dice", producing a random number between 1 and 6.

When I write code with effects, I usually break it into two phases. Phase one is about getting something on screen, just doing the bare minimum to have something to work from. Phase two is filling in details, gradually approaching the actual goal. We will use this process here too.

Phase One - The Bare Minimum

As always, you start out by guessing at what your `Model` should be:

```
type alias Model =
  { dieFace : Int
  }
```

For now we will just track `dieFace` as an integer between 1 and 6. Then I would quickly sketch out the `view` function because it seems like the easiest next step.

```
view : Model -> Html Msg
view model =
  div []
    [ h1 [] [ text (toString model.dieFace) ]
    , button [ onClick Roll ] [ text "Roll" ]
    ]
```

So this is typical. Same stuff we have been doing with the user input examples of The Elm Architecture. When you click our `<button>` it is going to produce a `Roll` message, so I guess it is time to take a first pass at the `update` function as well.

```
type Msg = Roll

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Roll ->
      (model, Cmd.none)
```

Now the `update` function has the same overall shape as before, but the return type is a bit different. Instead of just giving back a `Model`, it produces both a `Model` and a command. The idea is: **we still want to step the model forward, but we also want to do some stuff.** In our case, we want to ask Elm to give us a random value. For now, I just fill it in with `Cmd.none` which means "I have no commands, do nothing." We will fill this in with the good stuff in phase two.

Finally, I would create an `init` value like this:

```
init : (Model, Cmd Msg)
init =
  (Model 1, Cmd.none)
```

Here we specify both the initial model and some commands we'd like to run immediately when the app starts. This is exactly the kind of stuff that `update` is producing now too.

At this point, it is possible to wire it all up and take a look. You can click the `<button>`, but nothing happens. Let's fix that!

Phase Two - Adding the Cool Stuff

The obvious thing missing right now is the randomness! When the user clicks a button we want to command Elm to reach into its internal random number generator and give us a number between 1 and 6. The first step I would take towards that goal would be adding a new kind of message:

```
type Msg
  = Roll
  | NewFace Int
```

We still have `Roll` from before, but now we add `NewFace` for when Elm hands us our new random number. That is enough to start filling in `update`:

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Roll ->
      (model, Random.generate NewFace (Random.int 1 6))

    NewFace newFace ->
      (Model newFace, Cmd.none)
```

There are two new things here. **First**, there is now a branch for `NewFace` messages. When a `NewFace` comes in, we just step the model forward and do nothing. **Second**, we have added a real command to the `Roll` branch. This uses a couple functions from the `Random` library. Most important is `Random.generate`:

```
Random.generate : (a -> msg) -> Random.Generator a -> Cmd msg
```

This function takes two arguments. The first is a function to tag random values. In our case we want to use `NewFace : Int -> Msg` to turn the random number into a message for our `update` function. The second argument is a "generator" which is like a recipe for producing certain types of random values. You can have generators for simple types like `Int` or `Float` or `Bool`, but also for fancy types like big custom records with lots of fields. In this case, we use one of the simplest generators:

```
Random.int : Int -> Int -> Random.Generator Int
```

You provide a lower and upper bound on the integer, and now you have a generator that produces integers in that range!

That is it. Now we can click and see the number flip to some new value!

So the big lessons here are:

- Write your programs bit by bit. Start with a simple skeleton, and gradually add the tougher stuff.
- The `update` function now produces a new model *and* a command.
- You cannot just get random values willy-nilly. You create a command, and Elm will go do some work behind the scenes to provide it for you. In fact, any time our program needs to get unreliable values (randomness, HTTP, file I/O, database reads, etc.) you have to go through Elm.

At this point, the best way to improve your understanding of commands is just to see more of them! They will appear prominently with the `Http` and `WebSocket` libraries, so if you are feeling shaky, the only path forward is practicing with randomness and playing with other

examples of commands!

> **Exercises:** Here are some that build on stuff that has already been introduced:
>
> - Instead of showing a number, show the die face as an image.
> - Add a second die and have them both roll at the same time.
>
> And here are some that require new skills:
>
> - Instead of showing an image of a die face, use the `elm-lang/svg` library to draw it yourself.
> - After you have learned about tasks and animation, have the dice flip around randomly before they settle on a final value.

HTTP

Clone the code or follow along in the online editor.

We are about to make an app that fetches a random GIF when the user asks for another image.

Now, I am going to assume you just read the randomness example. It (1) introduces a two step process for writing apps like this and (2) shows the simplest kind of commands possible. Here we will be using the same two step process to build up to fancier kinds of commands, so I very highly recommend going back one page. I swear you will reach your goals faster if you start with a solid foundation!

...

Okay, so you read it now right? Good. Let's get started on our random gif fetcher!

Phase One - The Bare Minimum

At this point in this guide, you should be pretty comfortable smacking down the basic skeleton of an Elm app. Guess at the model, fill in some messages, etc. etc.

```
-- MODEL

type alias Model =
  { topic : String
  , gifUrl : String
  }

init : (Model, Cmd Msg)
init =
  (Model "cats" "waiting.gif", Cmd.none)


-- UPDATE

type Msg = MorePlease

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    MorePlease ->
      (model, Cmd.none)


-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ h2 [] [text model.topic]
    , img [src model.gifUrl] []
    , button [ onClick MorePlease ] [ text "More Please!" ]
    ]
```

For the model, I decided to track a `topic` so I know what kind of gifs to fetch. I do not want to hard code it to `"cats"`, and maybe later we will want to let the user decide the topic too. I also tracked the `gifUrl` which is a URL that points at some random gif.

Like in the randomness example, I just made dummy `init` and `update` functions. None of them actually produce any commands for now. The point is just to get something on screen!

Phase Two - Adding the Cool Stuff

Alright, the obvious thing missing right now is the HTTP request. I think it is easiest to start this process by adding new kinds of messages. Now remember, **when you give a command, you have to wait for it to happen.** So when we command Elm to do an HTTP

request, it is eventually going to tell you "hey, here is what you wanted" or it is going to say "oops, something went wrong with the HTTP request". We need this to be reflected in our messages:

```
type Msg
  = MorePlease
  | NewGif (Result Http.Error String)
```

We add a case for `NewGif` that holds a `Result`. You can read more about results here, but the key idea is that it captures the two possible outcomes of an HTTP request. It either (1) succeeded with the URL of a random gif or (2) failed with some HTTP error (server is down, bad URL, etc.)

That is enough to start filling in `update`:

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    MorePlease ->
      (model, getRandomGif model.topic)

    NewGif (Ok newUrl) ->
      ( { model | gifUrl = newUrl }, Cmd.none)

    NewGif (Err _) ->
      (model, Cmd.none)
```

So I added branches for our new messages. When `NewGif` holds a success, we update the `gifUrl` field to have the new URL. When `NewGif` holds an error, we ignore it, giving back the same model and doing nothing.

I also changed the `MorePlease` branch a bit. We need an HTTP command, so I called the `getRandomGif` function. The trick is that I made that function up. It does not exist yet. That is the next step!

Defining `getRandomGif` might look something like this:

```
getRandomGif : String -> Cmd Msg
getRandomGif topic =
  let
    url =
      "https://api.giphy.com/v1/gifs/random?api_key=dc6zaTOxFJmzC&tag=" ++ topic

    request =
      Http.get url decodeGifUrl
  in
    Http.send NewGif request

decodeGifUrl : Decode.Decoder String
decodeGifUrl =
  Decode.at ["data", "image_url"] Decode.string
```

With that added, the "More" button actually goes and fetches a random gif. Check it out here! But how does `getRandomGif` work exactly?

It starts out simple, we define `url` to be some giphy endpoint for random gifs. Next we create an HTTP `request` with `Http.get`. Finally, we turn it into a command with `Http.send`. Let's break those steps down a bit more:

- `Http.get : String -> Decode.Decoder value -> Http.Request value`

  This function takes a URL and a JSON decoder. This is our first time seeing a JSON decoder (and we will cover them in depth later), but for now, you really just need a high-level understanding. It turns JSON into Elm. In our case, we defined `decodeGifUrl` which tries to find a string value at `json.data.image_url`. Between the URL and the JSON decoder, we create an `Http.Request`. This is similar to a `Random.Generator` like we saw in the previous example. It does not actually *do* anything. It just describes how to make an HTTP request.

- `Http.send : (Result Error value -> msg) -> Http.Request value -> Cmd msg`

  Once we have an HTTP request, we can turn it into a command with `Http.send`. This is just like how we used `Random.generate` to create a command with a random generator. The first argument is a way to turn the result of the HTTP request into a message for our `update` function. In this case, we create a `NewGif` message.

This has been a very quick introduction, but the key idea is that you must (1) create an HTTP request and (2) turn that into a command so Elm will actually *do* it. You can go pretty far using the basic pattern here, and we will be looking into JSON decoders later on, which will let you deal with whatever crazy JSON you run into.

> **Exercises:** To get more comfortable with this code, try augmenting it with skills we learned in previous sections:
>
> - Show a message explaining why the image didn't change when you get an `Http.Error`.
> - Allow the user to modify the `topic` with a text field.
> - Allow the user to modify the `topic` with a drop down menu.

Time

[Clone the code] or follow along in the [online editor].

We are going to make a simple clock.

So far we have focused on commands. With the randomness example, we *asked* for a random value. With the HTTP example, we *asked* for info from a server. That pattern does not really work for a clock. In this case, we want to sit around and hear about clock ticks whenever they happen. This is where **subscriptions** come in.

The code is not too crazy here, so I am going to include it in full. After you read through, we will come back to normal words that explain it in more depth.

```
import Html exposing (Html)
import Svg exposing (..)
import Svg.Attributes exposing (..)
import Time exposing (Time, second)



main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }


-- MODEL

type alias Model = Time


init : (Model, Cmd Msg)
init =
  (0, Cmd.none)


-- UPDATE

type Msg
  = Tick Time
```

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Tick newTime ->
      (newTime, Cmd.none)


-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every second Tick


-- VIEW

view : Model -> Html Msg
view model =
  let
    angle =
      turns (Time.inMinutes model)

    handX =
      toString (50 + 40 * cos angle)

    handY =
      toString (50 + 40 * sin angle)
  in
    svg [ viewBox "0 0 100 100", width "300px" ]
      [ circle [ cx "50", cy "50", r "45", fill "#0B79CE" ] []
      , line [ x1 "50", y1 "50", x2 handX, y2 handY, stroke "#023963" ] []
      ]
```

There is nothing new in the `MODEL` or `UPDATE` sections. Same old stuff. The `view` function is kind of interesting. Instead of using HTML, we use the `Svg` library to draw some shapes. It works just like HTML though. You provide a list of attributes and a list of children for every node.

The important thing comes in `SUBSCRIPTIONS` section. The `subscriptions` function takes in the model, and instead of returning `Sub.none` like in the examples we have seen so far, it gives back a real life subscription! In this case `Time.every` :

```
Time.every : Time -> (Time -> msg) -> Sub msg
```

The first argument is a time interval. We chose to get ticks every second. The second argument is a function that turns the current time into a message for the `update` function. We are tagging times with `Tick` so the time 1458863979862 would become `Tick 1458863979862`.

That is all there is to setting up a subscription! These messages will be fed to your `update` function whenever they become available.

> **Exercises:**
>
> - Add a button to pause the clock, turning the `Time` subscription off.
> - Make the clock look nicer. Add an hour and minute hand. Etc.

Web Sockets

Clone the code or follow along in the online editor.

We are going to make a simple chat app. There will be a text field so you can type things in and a region that shows all the messages we have received so far. Web sockets are great for this scenario because they let us set up a persistent connection with the server. This means:

1. You can send messages cheaply whenever you want.
2. The server can send *you* messages whenever it feels like it.

In other words, `WebSocket` is one of the rare libraries that makes use of both commands and subscriptions.

This program happens to be pretty short, so here is the full thing:

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import WebSocket


main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }


-- MODEL

type alias Model =
  { input : String
  , messages : List String
  }


init : (Model, Cmd Msg)
init =
  (Model "" [], Cmd.none)
```

```
-- UPDATE

type Msg
  = Input String
  | Send
  | NewMessage String


update : Msg -> Model -> (Model, Cmd Msg)
update msg {input, messages} =
  case msg of
    Input newInput ->
      (Model newInput messages, Cmd.none)

    Send ->
      (Model "" messages, WebSocket.send "ws://echo.websocket.org" input)

    NewMessage str ->
      (Model input (str :: messages), Cmd.none)


-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  WebSocket.listen "ws://echo.websocket.org" NewMessage


-- VIEW

view : Model -> Html Msg
view model =
  div []
    [ div [] (List.map viewMessage model.messages)
    , input [onInput Input] []
    , button [onClick Send] [text "Send"]
    ]


viewMessage : String -> Html msg
viewMessage msg =
  div [] [ text msg ]
```

The interesting parts are probably the uses of `WebSocket.send` and `WebSocket.listen`.

For simplicity we will target a simple server that just echos back whatever you type. So you will not be able to have the most exciting conversations in the basic version, but that is why we have exercises on these examples!

More about The Elm Architecture

The emphasis of this section has been: **how can we get people making cool Elm projects as quickly and smoothly as possible?** So we covered:

- The basic architecture pattern.
- How to create buttons and text fields.
- How to make HTTP requests.
- How to work with web sockets.

You can go quite far with this knowledge, but there are many important aspects of Elm itself that we have not covered yet. For example, union types are one of the most important features in the whole language and we have not focused on them at all!

So we are going to take a break from The Elm Architecture for a couple chapters to get a better understanding of Elm itself. We will come back to The Elm Architecture in a few chapters and focus on code reuse in larger applications. In the meantime, when a function gets so big it feels unmanageable in practice, make a helper function! Elm makes refactoring easy, so it is best to improve architecture as needed rather than preemptively. More about that later though!

P.S. Best not to skip ahead. You can build a bigger house if you have a strong foundation!

Types

One of Elm's major benefits is that **users do not see runtime errors in practice**. This is possible because the Elm compiler can analyze your source code very quickly to see how values flow through your program. If a value can ever be used in an invalid way, the compiler tells you about it with a friendly error message. This is called *type inference*. The compiler figures out what *type* of values flow in and out of all your functions.

An Example of Type Inference

The following code defines a `toFullName` function which extracts a persons full name as a string:

```
toFullName person =
  person.firstName ++ " " ++ person.lastName

fullName =
  toFullName { fistName = "Hermann", lastName = "Hesse" }
```

Like in JavaScript or Python, we just write the code with no extra clutter. Do you see the bug though?

In JavaScript, the equivalent code spits out `"undefined Hesse"`. Not even an error! Hopefully one of your users will tell you about it when they see it in the wild. In contrast, the Elm compiler just looks at the source code and tells you:

```
-- TYPE MISMATCH ---------------------------------------------------------------

The argument to function `toFullName` is causing a mismatch.

6│   toFullName { fistName = "Hermann", lastName = "Hesse" }
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Function `toFullName` is expecting the argument to be:

    { …, firstName : … }

But it is:

    { …, fistName : … }

Hint: I compared the record fields and found some potential typos.

    firstName <-> fistName
```

It sees that `toFullName` is getting the wrong *type* of argument. Like the hint in the error message says, someone accidentally wrote `fist` instead of `first`.

It is great to have an assistant for simple mistakes like this, but it is even more valuable when you have hundreds of files and a bunch of collaborators making changes. No matter how big and complex things get, the Elm compiler checks that *everything* fits together properly just based on the source code.

The better you understand types, the more the compiler feels like a friendly assistant. So let's start learning more!

Reading Types

In the Core Language section of this book, we ran a bunch of code in the REPL. Well, we are going to do it again, but now with an emphasis on the types that are getting spit out. So type `elm repl` in your terminal again. You should see this:

```
---- elm repl 0.18.0 -----------------------------------------------------------
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
>
```

Primitives and Lists

Let's enter some simple expressions and see what happens:

```
> "hello"
"hello" : String

> not True
False : Bool

> round 3.1415
3 : Int
```

In these three examples, the REPL tells us the resulting value along with what *type* of value it happens to be. The value `"hello"` is a `String`. The value `3` is an `Int`. Nothing too crazy here.

Let's see what happens with lists holding different types of values:

```
> [ "Alice", "Bob" ]
[ "Alice", "Bob" ] : List String

> [ 1.0, 8.6, 42.1 ]
[ 1.0, 8.6, 42.1 ] : List Float

> []
[] : List a
```

In the first case, we have a `List` filled with `String` values. In the second, the `List` is filled with `Float` values. In the third case the list is empty, so we do not actually know what kind of values are in the list. So the type `List a` is saying "I know I have a list, but it could be filled with anything". The lower-case `a` is called a *type variable*, meaning that there are no constraints in our program that pin this down to some specific type. In other words, the type can vary based on how it is used.

Functions

Let's see the type of some functions:

```
> import String
> String.length
<function> : String -> Int
```

The function `String.length` has type `String -> Int`. This means it *must* take in a `String` argument, and it will definitely return an integer result. So let's try giving it an argument:

```
> String.length "Supercalifragilisticexpialidocious"
34 : Int
```

The important thing to understand here is how the type of the result `Int` is built up from the initial expression. We have a `String -> Int` function and give it a `String` argument. This results in an `Int`.

What happens when you do not give a `String` though?

```
> String.length [1,2,3]
-- error!

> String.length True
-- error!
```

A `String -> Int` function *must* get a `String` argument!

Anonymous Functions

Elm has a feature called *anonymous functions*. Basically, you can create a function without naming it, like this:

```
> \n -> n / 2
<function> : Float -> Float
```

Between the backslash and the arrow, you list the arguments of the function, and on the right of the arrow, you say what to do with those arguments. In this example, it is saying: I take in some argument I will call `n` and then I am going to divide it by two.

We can use anonymous functions directly. Here is us using our anonymous function with `128` as the argument:

```
> (\n -> n / 2) 128
64 : Float
```

We start with a `Float -> Float` function and give it a `Float` argument. The result is another `Float`.

> **Notes:** The backslash that starts an anonymous function is supposed to look like a lambda `λ` if you squint. This is a possibly ill-conceived wink to the intellectual history that led to languages like Elm.
>
> Also, when we wrote the expression `(\n -> n / 2) 128`, it is important that we put parentheses around the anonymous function. After the arrow, Elm is just going to keep reading code as long as it can. The parentheses put bounds on this, indicating where the function body ends.

Named Functions

In the same way that we can name a value, we can name an anonymous function. So rebellious!

```
> oneHundredAndTwentyEight = 128.0
128 : Float

> half = \n -> n / 2
<function> : Float -> Float

> half oneHundredAndTwentyEight
64 : Float
```

In the end, it works just like when nothing was named. You have a `Float -> Float` function, you give it a `Float`, and you end up with another `Float`.

Here is the crazy secret though: this is how all functions are defined! You are just giving a name to an anonymous function. So when you see things like this:

```
> half n = n / 2
<function> : Float -> Float
```

You can think of it as a convenient shorthand for:

```
> half = \n -> n / 2
<function> : Float -> Float
```

This is true for all functions, no matter how many arguments they have. So now let's take that a step farther and think about what it means for functions with *multiple* arguments:

```
> divide x y = x / y
<function> : Float -> Float -> Float

> divide 3 2
1.5 : Float
```

That seems fine, but why are there *two* arrows in the type for `divide`?! To start out, it is fine to think that "all the arguments are separated by arrows, and whatever is last is the result of the function". So `divide` takes two arguments and returns a `Float`.

To really understand why there are two arrows in the type of `divide`, it helps to convert the definition to use anonymous functions.

```
> divide x y = x / y
<function> : Float -> Float -> Float

> divide x = \y -> x / y
<function> : Float -> Float -> Float

> divide = \x -> (\y -> x / y)
<function> : Float -> Float -> Float
```

All of these are totally equivalent. We just moved the arguments over, turning them into anonymous functions one at a time. So when we run an expression like `divide 3 2` we are actually doing a bunch of evaluation steps:

```
divide 3 2
(divide 3) 2                 -- Step 1 - Add the implicit parentheses
((\x -> (\y -> x / y)) 3) 2  -- Step 2 - Expand `divide`
(\y -> 3 / y) 2              -- Step 3 - Replace x with 3
3 / 2                        -- Step 4 - Replace y with 2
1.5                          -- Step 5 - Do the math
```

After you expand `divide`, you actually provide the arguments one at a time. Replacing `x` and `y` are actually two different steps.

Let's break that down a bit more to see how the types work. In evaluation step #3 we saw the following function:

```
> (\y -> 3 / y)
<function> : Float -> Float
```

It is a `Float -> Float` function, just like `half`. Now in step #2 we saw a fancier function:

```
> (\x -> (\y -> x / y))
<function> : Float -> Float -> Float
```

Well, we are starting with `\x -> ...` so we know the type is going to be something like `Float -> ...`. We also know that `(\y -> x / y)` has type `Float -> Float`.

So if you actually wrote down all the parentheses in the type, it would instead say **`Float -> (Float -> Float)`**. You provide arguments one at a time. So when you replace `x`, the result is actually *another function*.

It is the same with all functions in Elm:

```
> import String
> String.repeat
<function> : Int -> String -> String
```

This is really `Int -> (String -> String)` because you are providing the arguments one at a time.

Because all functions in Elm work this way, you do not need to give all the arguments at once. It is possible to say things like this:

```
> divide 128
<function> : Float -> Float

> String.repeat 3
<function> : String -> String
```

This is called *partial application*. It lets us use the `|>` operator to chain functions together in a nice way, and it is why function types have so many arrows!

Type Annotations

So far we have just let Elm figure out the types, but it also lets you write a *type annotation* on the line above a definition if you want. So when you are writing code, you can say things like this:

```
half : Float -> Float
half n =
  n / 2

divide : Float -> Float -> Float
divide x y =
  x / y

askVegeta : Int -> String
askVegeta powerLevel =
  if powerLevel > 9000 then
    "It's over 9000!!!"

  else
    "It is " ++ toString powerLevel ++ "."
```

People can make mistakes in type annotations, so what happens if they say the wrong thing? Well, the compiler does not make mistakes, so it still figures out the type on its own. It then checks that your annotation matches the real answer. In other words, the compiler will always verify that all the annotations you add are correct.

> **Note:** Some folks feel that it is odd that the type annotation goes on the line above the actual definition. The reasoning is that it should be easy and noninvasive to add a type annotation *later*. This way you can turn a sloppy prototype into higher-quality code just by adding lines.

Type Aliases

The whole point of type aliases is to make your type annotations easier to read.

As your programs get more complicated, you find yourself working with larger and more complex data. For example, maybe you are making twitter-for-dogs and you need to represent a user. And maybe you want a function that checks to see if a user has a bio or not. You might write a function like this:

```
hasBio : { name : String, bio : String, pic : String } -> Bool
hasBio user =
  String.length user.bio > 0
```

That type annotation is kind of a mess, and users do not even have that many details! Imagine if there were ten fields. Or if you had a function that took users as an argument and gave users as the result.

In cases like this, you should create a *type alias* for your data:

```
type alias User =
  { name : String
  , bio : String
  , pic : String
  }
```

This is saying, wherever you see `User`, replace it by all this other stuff. So now we can rewrite our `hasBio` function in a much nicer way:

```
hasBio : User -> Bool
hasBio user =
  String.length user.bio > 0
```

Looks way better! It is important to emphasize that *these two definitions are exactly the same*. We just made an alias so we can say the same thing in fewer key strokes.

So if we write a function to add a bio, it would be like this:

```
addBio : String -> User -> User
addBio bio user =
  { user | bio = bio }
```

Imagine what that type annotation would look like if we did not have the `User` type alias. Bad!

Type aliases are not just about cosmetics though. They can help you think more clearly. When writing Elm programs, it is often best to *start* with the type alias before writing a bunch of functions. I find it helps direct my progress in a way that ends up being more efficient overall. Suddenly you know exactly what kind of data you are working with. If you need to add stuff to it, the compiler will tell you about any existing code that is affected by it. I think most experienced Elm folks use a similar process when working with records especially.

> **Note:** When you create a type alias specifically for a record, it also generates a *record constructor*. So our `User` type alias will also generate this function:
>
> ```
> User : String -> String -> String -> User
> ```
>
> The arguments are in the order they appear in the type alias declaration, so in the REPL you could do this:
>
> ```
> > type alias User = { name : String, bio : String, pic : String }
>
> > User "Tom" "Friendly Carpenter" "http://example.com/tom.jpg"
> { name = "Tom", bio = "Friendly Carpenter", pic = "http://example.com/tom.jpg" }
> : User
> ```
>
> This can be pretty handy!

Union Types

Many languages have trouble expressing data with weird shapes. They give you a small set of built-in types, and you have to represent everything with them. So you often find yourself using `null` or booleans or strings to encode details in a way that is quite error prone.

Elm's *union types* let you represent complex data much more naturally. We will go through a couple concrete examples to build some intuition about how and when to use union types.

> **Note:** Union types are sometimes called tagged unions. Some communities call them ADTs.

Filtering a Todo List

> **Problem:** We are creating a todo list full of tasks. We want to have three views: show *all* tasks, show only *active* tasks, and show only *completed* tasks. How do we represent which of these three states we are in?

Whenever you have weird shaped data in Elm, you want to reach for a union type. In this case, we would create a type `Visibility` that has three possible values:

```
> type Visibility = All | Active | Completed

> All
All : Visibility

> Active
Active : Visibility

> Completed
Completed : Visibility
```

Now that we have these three cases defined, we want to create a function `keep` that will properly filter our tasks. It should work like this:

```
type alias Task = { task : String, complete : Bool }

buy : Task
buy =
  { task = "Buy milk", complete = True }

drink : Task
drink =
  { task = "Drink milk", complete = False }

tasks : List Task
tasks =
  [ buy, drink ]


-- keep : Visibility -> List Task -> List Task

-- keep All tasks == [buy,drink]
-- keep Active tasks == [drink]
-- keep Complete tasks == [buy]
```

So the `keep` function needs to look at its first argument, and depending on what it is, filter the list in various ways. We use a `case` expression to do this. It is like an `if` on steroids:

```
keep : Visibility -> List Task -> List Task
keep visibility tasks =
  case visibility of
    All ->
      tasks

    Active ->
      List.filter (\task -> not task.complete) tasks

    Completed ->
      List.filter (\task -> task.complete) tasks
```

The `case` is saying, look at the structure of `visibility`. If it is `All`, just give back all the tasks. If it is `Active`, keep only the tasks that are not complete. If it is `Completed`, keep only the tasks that are complete.

The cool thing about `case` expressions is that all the branches are checked by the compiler. This has some nice benefits:

1. If you mistype `Compleet` by accident, you get a hint about the typo.
2. If you forget to handle a case, the compiler will figure it out and tell you.

So say you want to add `Recent` as a fourth possible `Visibility` value. The compiler will find all the `case` expressions in your code that work with `Visibility` values and remind you to handle the new possibility! This means you can change and extend `Visibility` without the risk of silently creating bugs in existing code.

> **Exercise:** Imagine how you would solve this same problem in JavaScript. Three strings? A boolean that can be `null`? What would the definition of `keep` look like? What sort of tests would you want to write to make sure adding new code later was safe.

Anonymous Users

> **Problem:** We have a chat room where people can post whatever they want. Some users are logged in and some are anonymous. How should we represent a user?

Again, whenever there is weird shaped data, you want to reach for a union type. For this case, we want one where users are either anonymous or named:

```
> type User = Anonymous | Named String

> Anonymous
Anonymous : User

> Named
<function> : String -> User

> Named "AzureDiamond"
Named "AzureDiamond" : User

> Named "abraham-lincoln"
Named "abraham-lincoln" : User
```

So creating the type `User` also created constructors named `Anonymous` and `Named`. If you want to create a `User` you *must* use one of these two constructors. This guarantees that all the possible `User` values are things like:

```
  Anonymous
  Named "AzureDiamond"
  Named "abraham-lincoln"
  Named "catface420"
  Named "Tom"
  ...
```

Now that we have a representation of a user, lets say we want to get a photo of them to show next to their posts. Again, we need to use a `case` expression to work with our `User` type:

```
userPhoto : User -> String
userPhoto user =
  case user of
    Anonymous ->
      "anon.png"

    Named name ->
      "users/" ++ name ++ ".png"
```

There are two possible cases when we have a `User`. If they are `Anonymous` we show a dummy picture. If they are `Named` we construct the URL of their photo. This `case` is slightly fancier than the one we saw before. Notice that the second branch has a lower case variable `name`. This means that when we see a value like `Named "AzureDiamond"`, the `name` variable will be bound to `"AzureDiamond"` so we can do other things with it. This is called *pattern matching*.

Now imagine we have a bunch of users in a chat room and we want to show their pictures.

```
activeUsers : List User
activeUsers =
  [ Anonymous, Named "catface420", Named "AzureDiamond", Anonymous ]

photos : List String
photos =
  List.map userPhoto activeUsers

-- [ "anon.png", "users/catface420.png", "users/AzureDiamond.png", "anon.png" ]
```

The nice thing about creating a type like `User` is that no one in your whole codebase can ever "forget" that some users may be anonymous. Anyone who can get a hold of a `User` needs to use a `case` to get any information out of it, and the compiler guarantees every `case` and handles all possible scenarios!

> **Exercise:** Think about how you would solve this problem in some other language. A string where empty string means they are anonymous? A string that can be null? How much testing would you want to do to make sure that everyone handles these special cases correctly?

Widget Dashboard

> **Problem:** You are creating a dashboard with three different kinds of widgets. One shows recent log data, one shows time plots, and one shows scatter plots. How do you represent a widget?

Alright, we are getting a bit fancier now. In Elm, you want to start by solving each case individually. (As you get more experience, you will see that Elm *wants* you to build programs out of small, reusable parts. It is weird.) So I would create representations for each of our three scenarios, along with `view` functions to actually turn them into HTML or SVG or whatever:

```
type alias LogsInfo =
  { logs : List String
  }

type alias TimeInfo =
  { events : List (Time, Float)
  , yAxis : String
  }

type alias ScatterInfo =
  { points : List (Float, Float)
  , xAxis : String
  , yAxis : String
  }

-- viewLogs : LogsInfo -> Html msg
-- viewTime : TimeInfo -> Html msg
-- viewScatter : ScatterInfo -> Html msg
```

At this point, you have created all the helper functions needed to work with these three cases totally independent from each other. Someone can come along later and say, "I need a nice way to show scatter plots" and use just that part of the code.

So the question is really: how do I put these three standalone things together for my particular scenario?

Again, union types are there to put together a bunch of different types!

```
> type Widget = Logs LogsInfo | TimePlot TimeInfo | ScatterPlot ScatterInfo

> Logs
<function> : LogsInfo -> Widget

> TimePlot
<function> : TimeInfo -> Widget

> ScatterPlot
<function> : ScatterInfo -> Widget
```

So we created a `Widget` type that can only be created with these constructor functions. You can think of these constructors as *tagging* the data so we can tell it apart at runtime. Now we can write something to render a widget like this:

```
view : Widget -> Html msg
view widget =
  case widget of
    Logs info ->
      viewLogs info

    TimePlot info ->
      viewTime info

    ScatterPlot info ->
      viewScatter info
```

One nice thing about this approach is that there is no mystery about what kind of widgets are supported. There are exactly three. If someone wants to add a fourth, they modify the `Widget` type. This means you can never be surprised by the data you get, even if someone on a different team is messing with your code.

> **Takeaways:**
>
> - Solve each subproblem first.
> - Use union types to put together all the solutions.
> - Creating a union type generates a bunch of *constructors*.
> - These constuctors *tag* data so that we can differentiate it at runtime.
> - A `case` expression lets us tear data apart based on these tags.
>
> The same strategies can be used if you are making a game and have a bunch of different bad guys. Goombas should update one way, but Koopa Troopas do something totally different. Solve each problem independently, and then use a union type to put them all together.

Linked Lists

> **Problem:** You are stuck on a bus speeding down the highway. If the bus slows down, it will blow up. The only way to save yourself and everyone on the bus is to reimplement linked lists in Elm. HURRY, WE ARE RUNNING OUT OF GAS!

Yeah, yeah, the problem is contrived this time, but it is important to see some of the more advanced things you can do with union types!

A linked list is a sequence of values. If you are looking at a linked list, it is either empty or it is a value and more list. That list is either empty or is a value and more list. etc. This intuitive definition works pretty directly in Elm. Let's see it for lists of integers:

```
> type IntList = Empty | Node Int IntList

> Empty
Empty : IntList

> Node
<function> : Int -> IntList -> IntList

> Node 42 Empty
Node 42 Empty : IntList

> Node 64 (Node 128 Empty)
Node 64 (Node 128 Empty) : IntList
```

Now we did two new things here:

1. The `Node` constructor takes *two* arguments instead of one. This is fine. In fact, you can have them take as many arguments as you want.
2. Our union type is *recursive*. An `IntList` may hold another `IntList`. Again, this is fine if you are using union types.

The nice thing about our `IntList` type is that now we can only build valid linked lists. Every linked list needs to start with `Empty` and the only way to add a new value is with `Node`.

It is equally nice to work with. Let's say we want to compute the sum of all of the numbers in a list. Just like with any other union type, we need to use a `case` and handle all possible scenarios:

```
sum : IntList -> Int
sum numbers =
  case numbers of
    Empty ->
      0

    Node n remainingNumbers ->
      n + sum remainingNumbers
```

If we get an `Empty` value, the sum is 0. If we have a `Node` we add the first element to the sum of all the remaining ones. So an expression like `(sum (Node 1 (Node 2 (Node 3 Empty))))` is evaluated like this:

```
  sum (Node 1 (Node 2 (Node 3 Empty)))
  1 + sum (Node 2 (Node 3 Empty))
  1 + (2 + sum (Node 3 Empty))
  1 + (2 + (3 + sum Empty))
  1 + (2 + (3 + 0))
  1 + (2 + 3)
  1 + 5
  6
```

On each line, we see one evaluation step. When we call `sum` it transforms the list based on whether it is looking at a `Node` or an `Empty` value.

> **Note:** This is the first recursive function we have written together! Notice that `sum` calls itself to get the sum. It can be tricky to get into the mindset of writing recursive functions, so I wanted to share one weird trick. **Pretend you are already done.**
>
> I always start with a `case` and all of the branches listed but not filled in. From there, I solve each branch one at a time, pretending that nothing else exists. So with `sum` I'd look at `Empty ->` and say, an empty list has to sum to zero. Then I'd look at the `Node n remainingNumbers ->` branch and think, well, I know I have a number, a list, and a `sum` function that definitely already exists and totally works. I can just use that and add a number to it!

Generic Data Structures

> **Problem:** The last section showed linked lists that only worked for integers. That is pretty lame. How can we make linked lists that hold any kind of value?

Everything is going to be pretty much the same, except we are going to introduce a *type variable* in our definition of lists:

```
> type List a = Empty | Node a (List a)

> Empty
Empty : List a

> Node
<function> : a -> List a -> List a

> Node "hi" Empty
Node "hi" Empty : List String

> Node 1.618 (Node 6.283 Empty)
Node 1.618 (Node 6.283 Empty) : List Float
```

The fancy part comes in the `Node` constructor. Instead of pinning the data to `Int` and `IntList`, we say that it can hold `a` and `List a`. Basically, you can add a value as long as it is the same type of value as everything else in the list.

Everything else is the same. You pattern match on lists with `case` and you write recursive functions. The only difference is that our lists can hold anything now!

> **Exercise:** This is exactly how the `List` type in Elm works, so take a look at the `List` library and see if you can implement some of those functions yourself.

Additional Examples

We have seen a couple scenarios, but the best way to get more comfortable is to use union types more! So here are two examples that are kind of fun.

Binary Trees

Binary trees are almost exactly the same as linked lists:

```
> type Tree a = Empty | Node a (Tree a) (Tree a)

> Node
<function> : a -> Tree a -> Tree a -> Tree a

> Node "hi" Empty Empty
Node "hi" Empty Empty : Tree String
```

A tree is either empty or it is a node with a value and two children. Check out this example for more info on this. If you can do all of the exercises at the end of that link, consider yourself a capable user of union types!

Languages

We can even model a programming language as data if we want to go really crazy! In this case, it is one that only deals with Boolean algebra:

```
type Boolean
    = T
    | F
    | Not Boolean
    | And Boolean Boolean
    | Or Boolean Boolean

true = Or T F
false = And T (Not T)
```

Once we have modeled the possible values we can define functions like `eval` which evaluates any `Boolean` to `True` or `False`. See this example for more about representing boolean expressions.

Error Handling and Tasks

One of the guarantees of Elm is that you will not see runtime errors in practice. NoRedInk has been using Elm in production for about a year now, and they still have not had one! Like all guarantees in Elm, this comes down to fundamental language design choices. In this case, we are helped by the fact that **Elm treats errors as data.** (Have you noticed we make things data a lot here?)

This section is going to walk through three data structures that help you handle errors in a couple different ways.

- `Maybe`
- `Result`
- `Task`

Now some of you probably want to jump right to tasks, but trust me that going in order will help here! You can think of these three data structures as a progression that slowly address crazier and crazier situations. So if you jump in at the end, it will be a lot to figure out all at once.

Some Historical Context

There are two popular language features that consistently cause unexpected problems. If you have used Java or C or JavaScript or Python or Ruby, you have almost certainly had your code crash because of `null` values or surprise exceptions from someone else's code.

Now these things are extremely familiar to folks, but that does not mean they are good!

Null

Any time you think you have a `String` you just might have a `null` instead. Should you check? Did the person giving you the value check? Maybe it will be fine? Maybe it will crash your servers? I guess we will find out later!

The inventor, Tony Hoare, has this to say about it:

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

As we will see soon, the point of `Maybe` is to avoid this problem in a pleasant way.

Exceptions

Joel Spolsky outlined the issues with exceptions pretty nicely in the year 2003. Essentially, code that *looks* fine may actually crash at runtime. Surprise!

The point of `Result` is to make the possibility of failure clear and make sure it is handled appropriately.

The point of `Task` is pretty much the same, but it also works when we have code that runs asynchronously. Your error handling mechanism shouldn't totally fall apart just because you make an HTTP request!

Maybe

It is best to just start with the definition of `Maybe` . It is a union type just like in all the examples here. It is defined like this:

```
> type Maybe a = Nothing | Just a

> Nothing
Nothing : Maybe a

> Just
<function> : a -> Maybe a

> Just "hello"
Just "hello" : Maybe String

> Just 1.618
Just 1.618 : Maybe Float
```

If you want to have a `Maybe` value, you have to use the `Nothing` or `Just` constructors to create it. This means that to deal with the data, you have to use a `case` expression. This means the compiler can ensure that you have definitely covered both possibilities!

There are two major cases where you will see `Maybe` values.

Optional Fields

Say we are running a social networking website. Connecting people, friendship, etc. You know the spiel. The Onion outlined our real goals best: mine as much data as possible for the CIA. And if we want *all* the data, we need to ease people into it. Let them add it later. Add features that encourage them to share more and more information over time.

So let's start with a simple model of a user. They must have a name, but we are going to make the age optional.

```
type alias User =
  { name : String
  , age : Maybe Int
  }
```

Now say Sue logs in and decides not to provide her birthday:

```
sue : User
sue =
  { name = "Sue", age = Nothing }
```

Now her friends cannot wish her a happy birthday. Sad! Later Tom creates a profile and *does* give his age:

```
tom : User
tom =
  { name = "Tom", age = Just 24 }
```

Great, that will be nice on his birthday. But more importantly, Tom is part of a valuable demographic! The advertisers will be pleased.

Alright, so now that we have some users, how can we market alcohol to them without breaking any laws? People would probably be mad if we market to people under 21, so let's check for that:

```
canBuyAlcohol : User -> Bool
canBuyAlcohol user =
  case user.age of
    Nothing ->
      False

    Just age ->
      age >= 21
```

Now the cool thing is that we are forced to use a `case` to pattern match on the users age. It is actually impossible to write code where you forget that users may not have an age. Elm can make sure of it. Now we can advertise alcohol confident that we are not influencing minors directly! Only their older peers.

Partial Functions

Sometimes you want a function that gives an answer sometimes, but just does not in other cases.

Let's say Mountain Dew wants to do some ad buys for people ages 13 to 18. Honestly, it is better to start kids on Mountain Dew younger than that, but it is illegal for kids under 13 to be on our site.

So let's say we want to write a function that will tell us a user's age, but only if they are between 13 and 18:

```
getTeenAge : User -> Maybe Int
getTeenAge user =
  case user.age of
    Nothing ->
      Nothing

    Just age ->
      if 13 <= age && age <= 18 then
        Just age

      else
        Nothing
```

Again, we are reminded that users may not have an age, but if they do, we only want to return it if it is between 13 and 18. Now Elm can guarantee that anyone who calls `getTeenAge` will have to handle the possibility that the age is out of range.

This gets pretty cool when you start combining it with library functions like `List.filterMap` that help you process more data. For example, maybe we want to figure out the distribution of ages between 13 and 18. We could do it like this:

```
> alice = User "Alice" (Just 14)
... : User

> bob = User "Bob" (Just 16)
... : User

> users = [ sue, tom, alice, bob ]
... : List User

> List.filterMap getTeenAge users
[14,16] : List Int
```

We end up with only the ages we care about. Now we can feed our `List Int` into a function that figures out the distributions of each number.

Result

A `Result` is useful when you have logic that may "fail". For example, parsing a `String` into an `Int` may fail. What if the string is filled with the letter B? In cases like this, we want a function with this type:

```
String.toInt : String -> Result String Int
```

This means that `String.toInt` will take in a string value and start processing the string. If it cannot be turned into an integer, we provide a `String` error message. If it can be turned into an integer, we return that `Int`. So the `Result String Int` type is saying, "my errors are strings and my successes are integers."

To make this as concrete as possible, let's see the actual definition of `Result`. It is actually pretty similar to the `Maybe` type, but it has *two* type variables:

```
type Result error value
  = Err error
  | Ok value
```

You have two constructors: `Err` to tag errors and `Ok` to tag successes. Let's see what happens when we actually use `String.toInt` in the REPL:

```
> import String

> String.toInt "128"
Ok 128 : Result String Int

> String.toInt "64"
Ok 64 : Result String Int

> String.toInt "BBBB"
Err "could not convert string 'BBBB' to an Int" : Result String Int
```

So instead of throwing an exception like in most languages, we return data that indicates whether things have gone well or not. Let's imagine someone is typing their age into a text field and we want to show a validation message:

```
view : String -> Html msg
view userInputAge =
  case String.toInt userInputAge of
    Err msg ->
      span [class "error"] [text msg]

    Ok age ->
      if age < 0 then
        span [class "error"] [text "I bet you are older than that!"]

      else if age > 140 then
        span [class "error"] [text "Seems unlikely..."]

      else
        text "OK!"
```

Again, we have to use `case` so we are guaranteed to handle the special case where the number is bad.

Tasks

These docs are getting updated for 0.18. They will be back soon! Until then, the docs will give a partial overview.

Interop

Interop is extraordinarily important if you want your language to succeed!

This is just a historical fact. A huge part of why C++ was so successful was that it was easy to migrate a massive C codebase. If you look at the JVM, you see Scala and Clojure carving out pretty big niches for themselves thanks to their nice interop story with Java. For industrial users, there is no point in having an amazing language with great guarantees if there is no way to slowly introduce it into an existing codebase. It is exactly the same in browsers too.

This section focuses on the major kinds of interop that you need when working in browsers.

1. How to communicate with external services using JSON.
2. How to embed Elm programs in existing HTML or React apps.
3. How to communicate with existing JavaScript code.

Each of these types of interop are guided by the self-imposed constraints that (1) there must be a clear way to introduce Elm gradually into diverse environments and (2) Elm should not have to sacrifice its core design principles. In other words, **Elm should be great *and* it should be possible to use Elm at work.**

Advice on Introducing Elm

The correct path is to first use Elm in a small experiment. If the experiment goes bad, stop it! If it goes great, expand the experiment a bit more. Then just repeat this process until you are using Elm or not! History seems to suggest that there is no realistic way to translate an existing project into a new language all at once. You have to evolve gradually!

Every company I know of that introduced Elm into an existing codebase did it gradually. You need to make sure it is worth it. You probably need to do some pairing or mentorship to get your teammates comfortable. You may even want to use React as a stepping stone if you are on something before that. Basically, anything you can do to minimize risk and make the process feel gradual will improve your odds. Now none of this is as fun as just switching, but it has the great benefit of actually working out in practice.

> This section uses `elm-repl`. If you did not install it already, you can use the online REPL.

JSON

You will be sending lots of JSON in your programs. You use the `Json.Decode` library to convert wild and crazy JSON into nicely structured Elm values.

The core concept for working with JSON is called a **decoder**. It decodes JSON values into Elm values. We will start out by looking at some very basic decoders and then look at how to put them together to handle more complex scenarios.

Primitive Decoders

Here are the type signatures for a couple primitive decoders:

```
string : Decoder String
int : Decoder Int
float : Decoder Float
bool : Decoder Bool
```

These become useful when paired with the `decodeString` function:

```
decodeString : Decoder a -> String -> Result String a
```

This means we can do stuff like this:

```
> import Json.Decode exposing (..)

> decodeString int "42"
Ok 42 : Result String Int

> decodeString float "3.14159"
Ok 3.14159 : Result String Float

> decodeString bool "true"
Ok True : Result String Bool

> decodeString int "true"
Err "Expecting an Int but instead got: true" : Result String Int
```

So our little decoders let us turn strings of JSON values into a `Result` telling us how the conversion went.

Now that we can handle the simplest JSON values, how can we deal with more complex things like arrays and objects?

Combining Decoders

The cool thing about decoders is that they snap together like building blocks. So if we want to handle a list of values, we would reach for the `list` function:

```
list : Decoder a -> Decoder (List a)
```

We can combine this with all the primitive decoders now:

```
> import Json.Decode exposing (..)

> int
<decoder> : Decoder Int

> list int
<decoder> : Decoder (List Int)

> decodeString (list int) "[1,2,3]"
Ok [1,2,3] : Result String (List Int)

> decodeString (list string) """["hi", "yo"]"""
Ok ["hi", "yo"] : Result String (List String)
```

So now we can handle JSON arrays. If we want to get extra crazy, we can even nest lists.

```
> decodeString (list (list int)) "[ [0], [1,2,3], [4,5] ]"
Ok [[0],[1,2,3],[4,5]] : Result String (List (List Int))
```

So that is `list`, but `Json.Decode` can handle many other data structures too. For example, `dict` helps you turn a JSON object into an Elm `Dict` and `keyValuePairs` helps you turn a JSON object into an Elm list of keys and values.

Decoding Objects

We decode JSON objects with the `field` function. It snaps together decoders just like `list`:

```
field : String -> Decoder a -> Decoder a
```

So when you say `field "x" int` you are saying (1) I want a JSON object, (2) it should have a field `x`, and (3) the value at `x` should be an integer. So using it looks like this:

```
> import Json.Decode exposing (..)

> field "x" int
<decoder> : Decoder Int

> decodeString (field "x" int) """{ "x": 3, "y": 4 }"""
Ok 3 : Result String Int

> decodeString (field "y" int) """{ "x": 3, "y": 4 }"""
Ok 4 : Result String Int
```

Notice that the `field "x" int` decoder only cares about field `x`. The object can have other fields with other content. That is all separate. But what happens when you want to get information from *many* fields? Well, we just need to put together many decoders. This is possible with functions like `map2`:

```
map2 : (a -> b -> value) -> Decoder a -> Decoder b -> Decoder value
```

This function takes in two different decoders. If they are both successful, it uses the given function to combine their results. So now we can put together two different decoders:

```
> import Json.Decode exposing (..)

> type alias Point = { x : Int, y : Int }

> Point
<function> : Int -> Int -> Point

> pointDecoder = map2 Point (field "x" int) (field "y" int)
<decoder> : Decoder Point

> decodeString pointDecoder """{ "x": 3, "y": 4 }"""
Ok { x = 3, y = 4 } : Result String Point
```

Okay, that covers two fields, but what about three? Or four? The core library provides `map3`, `map4`, and others for handling larger objects.

As you start working with larger JSON objects, it is worth checking out `NoRedInk/elm-decode-pipeline`. It builds on top of the core `Json.Decode` module described here and lets you write stuff like this:

```
import Json.Decode exposing (Decoder, int)
import Json.Decode.Pipeline exposing (decode, required)

type alias Point = { x : Int, y : Int }

pointDecoder : Decoder Point
pointDecoder =
  decode Point
    |> required "x" int
    |> required "y" int
```

You can have `optional` and `hardcoded` fields as well. It is quite a nice library, so take a look!

Broader Context

By now you have seen a pretty big chunk of the actual `Json.Decode` API, so I want to give some additional context about how this fits into the broader world of Elm and web apps.

Validating Server Data

The conversion from JSON to Elm doubles as a validation phase. You are not just converting from JSON, you are also making sure that JSON conforms to a particular structure.

In fact, decoders have revealed weird data coming from NoRedInk's *backend* code! If your server is producing unexpected values for JavaScript, the client just gradually crashes as you run into missing fields. In contrast, Elm recognizes JSON values with unexpected structure, so NoRedInk gives a nice explanation to the user and logs the unexpected value. This has actually led to some patches in their Ruby server!

A General Pattern

JSON decoders are an example of a more general pattern in Elm. You see it whenever you want to wrap up complicated logic into small building blocks that snap together easily. Other examples include:

- `Random` — The `Random` library has the concept of a `Generator`. So a `Generator Int` creates random integers. You start with primitive building blocks that generate random `Int` or `Bool`. From there, you use functions like `list` and `map` to build up generators for fancier types.
- `Easing` — The Easing library has the concept of an `Interpolation`. An `Interpolation Float` describes how to slide between two floating point numbers. You start with interpolations for primitives like `Float` or `Color`. The cool thing is that these interpolations compose, so you can build them up for much fancier types.

As of this writing, there is some early work on Protocol Buffers (binary data format) that uses the same pattern. In the end you get a nice composable API for converting between Elm values and binary!

JavaScript Interop

At some point your Elm program is probably going to need to talk to JavaScript. We do this by (1) embedding Elm in HTML and (2) sending messages back and forth between Elm and JavaScript:



This way we can have access to full power of JavaScript, the good and the bad, without giving up on all the things that are nice about Elm.

Step 1: Embed in HTML

Normally when you run the Elm compiler, it will give you an HTML file that sets everything up for you. So running this:

```
elm-make src/Main.elm
```

Will result in a `index.html` file that you can just open up and start using. To do fancier stuff, we want to compile to JavaScript, so we modify the command slightly:

```
elm-make src/Main.elm --output=main.js
```

Now the compiler will generate a JavaScript file that lets you initialize your program like this:

```
<div id="main"></div>
<script src="main.js"></script>
<script>
    var node = document.getElementById('main');
    var app = Elm.Main.embed(node);
    // Note: if your Elm module is named "MyThing.Root" you
    // would call "Elm.MyThing.Root.embed(node)" instead.
</script>
```

This is doing three important things:

1. We create a `<div>` that will hold the Elm program.
2. We load the JavaScript generated by the Elm compiler.
3. We grab the relevant node and initialize our Elm program in it.

So now we can set Elm up in any `<div>` we want. So if you are using React, you can create a component that just sets this kind of thing up. If you are using Angular or Ember or something else, it should not be too crazy either. Just take over a `<div>`.

The next section gets into how to get your Elm and JavaScript code to communicate with each other in a nice way.

Step 2: Talk to JavaScript

There are two major ways for Elm and JavaScript to talk to each other: **ports** and **flags**.

Ports

Say we have a nice Elm program and everything is going fine, but we want to use some JavaScript spell-checking library to get a feature done real quick. The final result is shown here, and we will walk through the most important parts here.

Okay, in Elm, **any communication with JavaScript goes through a port**. Think of it like a hole in the side of your Elm program where you can send values in and out. These work exactly like the commands and subscriptions from the Architecture section. Sending values out to JS is a command. Listening for values coming in from JS is a subscription. Pretty neat!

So if we want to talk to this spell-checking library, our Elm program will need these additional declarations:

```
port module Spelling exposing (..)

...

-- port for sending strings out to JavaScript
port check : String -> Cmd msg

-- port for listening for suggestions from JavaScript
port suggestions : (List String -> msg) -> Sub msg

...
```

Again, you can see the whole file here, but these are the important additions:

1. We change the `module` declaration to `port module`. This indicates that `port` declarations should be permitted. (Very few modules should have ports in them!)
2. We create a `check` port. On the Elm side, we can create commands like `check "badger"`, resulting in a `Cmd msg` that sends strings to the JS side.
3. We create a `suggestions` port. This one looks a bit fancier than the `check` port, but imagine that it is creating `Sub (List String)`. You are essentially subscribing to lists of strings sent into Elm from JS. So when the spell-checking library has a suggestion, it will send things through. Now, the type of `suggestions` is a bit fancier than that. You provide a function from `(List String -> msg)` so you can convert that list of strings to your `Msg` type immediately. This makes it easy to deal with in your `update` function, but it is just for convenience. The real point is to send a `List String` from JS into Elm.

Okay, so after you run `elm-make Spelling.elm --output=spelling.js` you embed it in HTML like this:

```
<div id="spelling"></div>
<script src="spelling.js"></script>
<script>
    var app = Elm.Spelling.fullscreen();

    app.ports.check.subscribe(function(word) {
        var suggestions = spellCheck(word);
        app.ports.suggestions.send(suggestions);
    });

    function spellCheck(word) {
        // have a real implementation!
        return [];
    }
</script>
```

Okay, so all the ports you declare in your Elm program will be available as fields of `app.ports`. In the code above, we access `app.ports.check` and `app.ports.suggestions`. They work like this:

- We can subscribe to `app.ports.check`. Every time Elm says to send a value out, we will call this JavaScript function.
- We can send values to `app.ports.suggestions`. So whenever we have some new suggestions for Elm, we just `send` them through.

With that knowledge, we can communicate back and forth with JavaScript!

> **Note:** Elm validates all values coming in from JavaScript. In Elm we said we can only handle `List String` so we need to make sure that the JavaScript code does not break that contract! More about that farther down this page.

Flags

The second way to talk to JavaScript is with *flags*. You can think of this as some static configuration for your Elm program.

Instead of creating a `Program` with the `program` function, we can use the `programWithFlags`. So say we want to get a value like this from JavaScript on initialization:

```
type alias Flags =
  { user : String
  , token : String
  }
```

We would set up our Elm program like this:

```
init : Flags -> ( Model, Cmd Msg )
init flags =
  ...

main =
  programWithFlags { init = init, ... }
```

And on the JavaScript side, we start the program like this:

```
// if you want it to be fullscreen
var app = Elm.MyApp.fullscreen({
    user: 'Tom',
    token: '12345'
});

// if you want to embed your app
var node = document.getElementById('my-app');
var app = Elm.MyApp.embed(node, {
    user: 'Tom',
    token: '12345'
});
```

Notice that this is exactly the same as normal, but we provide an extra argument with all the flags we want.

Just like ports, the values sent in from JavaScript are validated to make sure JavaScript bugs stay in JavaScript.

Customs and Border Protection

Ports and flags must be careful about what values are allowed through. Elm is statically typed, so each port is fitted with some border protection code that ensures that type errors are kept out. Ports also do some conversions so that you get nice colloquial data structures in both Elm and JS.

The particular types that can be sent in and out of ports are quite flexible, covering all valid JSON values. Specifically, incoming ports can handle all the following Elm types:

- **Booleans and Strings** – both exist in Elm and JS!
- **Numbers** – Elm ints and floats correspond to JS numbers
- **Lists** – correspond to JS arrays
- **Arrays** – correspond to JS arrays
- **Tuples** – correspond to fixed-length, mixed-type JS arrays
- **Records** – correspond to JavaScript objects
- **Maybes** – `Nothing` and `Just 42` correspond to `null` and `42` in JS
- **Json** – `Json.Encode.Value` corresponds to arbitrary JSON

Now say Elm wants a `List String`, but someone calls `app.ports.suggestions.send(42)` on the JavaScript side. We *know* it will cause issues in Elm, and we *know* the code producing invalid data is on the JS side. So rather than letting the bad data into Elm and cause a

runtime exception *eventually* (the JavaScript way!) we throw a runtime exception *immediately* when you call `send` with invalid data. So we cannot solve the problem of invalid data in JavaScript, but we can at least make sure it stays on the JavaScript side!

Usage Advice

I showed an example where the ports were declared in the root module. This is not a strict requirement. You can actually create a `port module` that gets imported by various parts of your app.

It seems like it is probably best to just have one `port module` for your project so it is easier to figure out the API on the JavaScript side. I plan to improve tooling such that you can just ask though.

> **Note:** Port modules are not permitted in the package repository. Imagine you download an Elm package and it just doesn't work. You read the docs and discover you *also* need to get some JS code and hook it up properly. Lame. Bad experience. Now imagine if you had this risk with *every* package out there. It just would feel crappy, so we do not allow that.

Historical Context

Now I know that this is not the typical interpretation of *language interop*. Usually languages just go for full backwards compatibility. So C code can be used *anywhere* in C++ code. You can replace C/C++ with Java/Scala or JavaScript/TypeScript. This is the easiest solution, but it forces you to make quite extreme sacrifices in the new language. All the problems of the old language now exist in the new one too. Hopefully less though.

Elm's interop is built on the observation that **by enforcing some architectural rules, you can make full use of the old language *without* making sacrifices in the new one.** This means we can keep making guarantees like "you will not see runtime errors in Elm" even as you start introducing whatever crazy JavaScript code you need.

So what are these architectural rules? Turns out it is just The Elm Architecture. Instead of embedding arbitrary JS code right in the middle of Elm, we use commands and subscriptions to send messages to external JavaScript code. So just like how the `WebSocket` library insulates you from all the crazy failures that might happen with web sockets, port modules insulate you from all the crazy failures that might happen in JavaScript. **It is like JavaScript-as-a-Service.**

Scaling The Elm Architecture

If you are coming from JavaScript, you are probably wondering “where are my reusable components?” and “how do I do parent-child communication between them?” A great deal of time and effort is spent on these questions in JavaScript, but it just works different in Elm. **We do not think in terms of reusable components.** Instead, we focus on reusable *functions*. It is a functional language after all!

So this chapter will go through a few examples that show how we create **reusable views** by breaking out helper functions to display our data. We will also learn about Elm’s *module system* which helps you break your code into multiple files and hide implementation details. These are the core tools and techniques of building large app with Elm.

In the end, I think we end up with something far more flexible and reliable than “reusable components” and there is no real trick. We will just be using the fundamental tools of functional programming!

Labeled Checkboxes

Follow along in the online editor.

Your app will probably have some options people can mess with. If something happens, should you send them an email notification? If they come across a video, should it start playing by itself? That kind of thing. So you will need to create some HTML like this:

```
<fieldset>
  <label><input type="checkbox">Email Notifications</label>
  <label><input type="checkbox">Video Autoplay</label>
  <label><input type="checkbox">Use Location</label>
</fieldset>
```

That will let people toggle the checkboxes, and using `<label>` means they get a much bigger area they can click on. Let's write an Elm program that manages all this interaction! As always, we will take a guess at our `Model`. We know we need to track the user's settings so we will put them in our model:

```
type alias Model =
  { notifications : Bool
  , autoplay : Bool
  , location : Bool
  }
```

From there, we will want to figure out our messages and update function. Maybe something like this:

```
type Msg
  = ToggleNotifications
  | ToggleAutoplay
  | ToggleLocation

update : Msg -> Model -> Model
update msg model =
  case msg of
    ToggleNotifications ->
      { model | notifications = not model.notifications }

    ToggleAutoplay ->
      { model | autoplay = not model.autoplay }

    ToggleLocation ->
      { model | location = not model.location }
```

That seems fine. Now to create our view!

```
view : Model -> Html Msg
view model =
  fieldset []
    [ label []
        [ input [ type_ "checkbox", onClick ToggleNotifications ] []
        , text "Email Notifications"
        ]
    , label []
        [ input [ type_ "checkbox", onClick ToggleAutoplay ] []
        , text "Video Autoplay"
        ]
    , label []
        [ input [ type_ "checkbox", onClick ToggleLocation ] []
        , text "Use Location"
        ]
    ]
```

This is not too crazy, but we are repeating ourselves a bit. How can we make our `view` function nicer? If you are coming from JavaScript, your first instinct is probably that we should make a “labeled checkbox component” but it is easier to just create a helper function! Here is the `view` function with a `checkbox` helper function:

```
view : Model -> Html Msg
view model =
  fieldset []
    [ checkbox ToggleNotifications "Email Notifications"
    , checkbox ToggleAutoplay "Video Autoplay"
    , checkbox ToggleLocation "Use Location"
    ]

checkbox : msg -> String -> Html msg
checkbox msg name =
  label []
    [ input [ type_ "checkbox", onClick msg ] []
    , text name
    ]
```

Now we have a highly configurable `checkbox` function. It takes two arguments to configure how it works: the message it produces on clicks and some text to show next to the checkbox. Now if we decide we want all checkboxes to have a certain `class`, we just add it in the `checkbox` function and it shows up everywhere! This is the essense of **reusable views** in Elm. Create helper functions!

Comparing Reusable Views to Reusable Components

We now have enough information to do a simple comparison of these approaches. Reusable views have a few major advantages over components:

- **It is just functions.** We are not doing anything special here. Functions have all the power we need, and they are very simple to create. It is the most basic building block of Elm!

- **No parent-child communication.** If we had made a “checkbox component” we would have to figure out how to synchronize the state in the checkbox component with our overall model. “That checkbox says notifications are on, but the model says they are off!” Maybe we need a Flux store now? By using functions instead, we are able to have reuse in our view *without* disrupting our `Model` or `update`. They work exactly the same as before, no need to touch them!

This means we can always create reusable `view` code without changing our overall architecture or introducing any fancy ideas. Just write smaller functions. That sounds nice, but let’s see some more examples to make sure it is true!

Radio Buttons

Follow along in the online editor.

Say you have a website that is primarily about reading, like this guide! You may want to have a way to choose between small, medium, and large fonts so your readers can customize it for their preferences. In that case, you will want some HTML like this:

```
<fieldset>
  <label><input type="radio">Small</label>
  <label><input type="radio">Medium</label>
  <label><input type="radio">Large</label>
</fieldset>
```

Just like in the checkbox example from the previous page, this will let people choose the one they want, and using `<label>` means they get a much bigger area they can click on. Like always, we start with our `Model`. This one is kind of interesting because we can use union types to make it very reliable!

```
type alias Model =
  { fontSize : FontSize
  , content : String
  }

type FontSize = Small | Medium | Large
```

This means there are exactly three possible font sizes: `Small`, `Medium`, and `Large`. It is impossible to have any other value in our `fontSize` field. If you are coming from JavaScript, you know their alternative is to use strings or numbers and just hope that there is never a typo or mistake. You *could* use values like that in Elm, but why open yourself up to bugs for no reason?!

> **Note:** You should always be looking for opportunities to use union types in your data. The best way to avoid invalid states is to make them impossible to represent in the first place!

Alright, now we need to `update` our model. In this case we just want to switch between font sizes as the user toggles the radio buttons:

```
type Msg
  = SwitchTo FontSize

update : Msg -> Model -> Model
update msg model =
  case msg of
    SwitchTo newFontSize ->
      { model | fontSize = newFontSize }
```

Now we need to describe how to show our `Model` on screen. First let's see the one where we put all our code in one function and repeat ourselves a bunch of times:

```
view : Model -> Html Msg
view model =
  div []
    [ fieldset []
        [ label []
            [ input [ type_ "radio", onClick (SwitchTo Small) ] []
            , text "Small"
            ]
        , label []
            [ input [ type_ "radio", onClick (SwitchTo Medium) ] []
            , text "Medium"
            ]
        , label []
            [ input [ type_ "radio", onClick (SwitchTo Large) ] []
            , text "Large"
            ]
        ]
    , section [] [ text model.content ]
    ]
```

That is kind of a mess! The best thing to do is to start making helper functions (not components!). We see some repetition in the radio buttons, so we will start there.

```
view : Model -> Html Msg
view model =
  div []
    [ fieldset []
        [ radio (SwitchTo Small) "Small"
        , radio (SwitchTo Medium) "Medium"
        , radio (SwitchTo Large) "Large"
        ]
    , section [] [ text model.content ]
    ]

radio : msg -> String -> Html msg
radio msg name =
  label []
    [ input [ type_ "radio", onClick msg ] []
    , text name
    ]
```

Our `view` function is quite a bit easier to read now. Great!

If that is the only chunk of radio buttons on your page, you are done. But perhaps you have a couple sets of radio buttons. For example, this guide not only lets you set font size, but also color scheme and whether you use a serif or sans-serif font. Each of those can be implemented as a set of radio buttons, so we could do a bit more refactoring, like this:

```
view : Model -> Html Msg
view model =
  div []
    [ viewPicker
        [ ("Small", SwitchTo Small)
        , ("Medium", SwitchTo Medium)
        , ("Large", SwitchTo Large)
        ]
    , section [] [ text model.content ]
    ]

viewPicker : List (String, msg) -> Html msg
viewPicker options =
  fieldset [] (List.map radio options)

radio : (String, msg) -> Html msg
radio (name, msg) =
  label []
    [ input [ type_ "radio", onClick msg ] []
    , text name
    ]
```

So if we want to let users choose a color scheme or toggle serifs, the `view` can reuse `viewPicker` for each case. If we do that, we may want to add additional arguments to the `viewPicker` function. If we want to be able to set a class on each `<fieldset>`, we could add an argument like this:

```
viewPicker : String -> List (String, msg) -> Html msg
viewPicker pickerClass options =
  fieldset [ class pickerClass ] (List.map radio options)
```

Or if we wanted even more flexibility, we could let people pass in whatever attributes they please, like this:

```
viewPicker : List (Attribute msg) -> List (String, msg) -> Html msg
viewPicker attributes options =
  fieldset attributes (List.map radio options)
```

And if we wanted even MORE flexibility, we could let people pass in attributes for each radio button too! There is really no end to what can be configured. You just add a bit more information to an argument.

Too Much Reuse?

In this case, we saw quite a few ways to write the same code. But which way is the *right* way to do it? A good rule to pick an API is **choose the absolute simplest thing that does everything you need**. Here are some scenarios that test this rule:

1. There is the only radio button thing on your page. In that case, just make them! Do not worry about making a highly configurable and reusable function for radio buttons. Refactoring is easy in Elm, so wait for a legitimate need before doing that work!

2. There are a couple radio button things on your page, all with the same styling. That is how the options on this guide look. This is a great case for sharing a view function. You may not even need to change any classes or add any custom attributes. If you do not need that, do not design for it! It is easy to add later.

3. There are a couple radio button things on your page, but they are very different. You could do an extremely flexible picker that lets you configure everything, but at some point, things that *look* similar are not actually similar enough for this to be worth it. So if you ever find yourself with tons of complex arguments configuring a view function, you may have overdone it on the reuse. I personally would prefer to have two chunks of *similar* view code that are both simple and easy to change than one chunk of view code that is complex and hard to understand.

Point is, there is no magic recipe here. The answer will depend on the particulars of your application, and you should always try to find the simplest approach. Sometimes that means sharing code. Sometimes it means writing *similar* code. It takes practice and experience to get good at this, so do not be afraid to experiment to find simpler ways!

Modules

In the last few sections, we learned how to create reusable views. Whenever you start seeing a pattern in your `view` code, you can break it out into a helper function. But so far, we have just been growing our files longer and longer. At some point this gets out of control though, we do not want to have 2000 line files!

So Elm has *modules* to help you grow your codebase in a nice way. On the most basic level, modules let you break your code into multiple files. Like everything else in Elm, you should only reach for a fancier tool when you feel you *need* it. So if you have a 400 line file and notice that a bunch of code is all related to showing radio buttons in a certain way, it may be a good idea to move all the relevant functions and types into their own module.

Before we get into the nuances of using modules *appropriately*, let's learn how to use them at all!

Defining Modules

Every module starts with a *module declaration*. So if I wanted to define my own version of the `Maybe` module, I might start with something like this:

```
module Optional exposing (..)

type Optional a = Some a | None

isNone : Optional a -> Bool
isNone optional =
  case optional of
    Some _ ->
      False

    None ->
      True
```

The new thing here is that first line. You can read it as "This module is named `Optional` and it exposes *everything* to people who use the module."

Exposing everything is fine for prototyping and exploration, but a serious project will want to make the exposed values explicit, like this:

```
module Optional exposing ( Optional(..), isNone )
```

Read this as "This module is named `Optional` and it exposes the `Optional` type, the `Some` and `None` constructors, and the `isNone` function to people who use the module." Now there is no reason to list *everything* that is defined, so later we will see how this can be used to hide implementation details.

> **Note:** If you forget to add a module declaration, Elm will use this one instead:
>
> ```
> module Main exposing (..)
> ```
>
> This makes things easier for beginners working in just one file. They should not be confronted with the module system on their first day!

Using Modules

Okay, we have our `Optional` module, but how do we use it? We can create `import` declarations at the top of files that describe which modules are needed. So if we wanted to make the "No shoes, no shirt, no service" policy explicit in code, we could write this:

```
import Optional

noService : Optional.Optional a -> Optional.Optional a -> Bool
noService shoes shirt =
  Optional.isNone shoes && Optional.isNone shirt
```

The `import Optional` line means you can use anything exposed by the module as long as you put `Optional.` before it. So in the `noService` function, you see `Optional.Optional` and `Optional.isNone`. These are called *qualified* names. Which `isNone` is it? The one from the `Optional` module! It says it right there in the code.

Generally, it is best to always use qualified names. In a project with twenty imports, it is extremely helpful to be able to quickly see where a value comes from.

That said, there are a few ways to customize an import that can come in handy.

As

You can use the `as` keyword to provide a shorter name. To stick with the `Optional` module, we could abbreviate it to just `Opt` like this:

```
import Optional as Opt

noService : Opt.Optional a -> Opt.Optional a -> Bool
noService shoes shirt =
  Opt.isNone shoes && Opt.isNone shirt
```

Now we can refer to `Opt.Optional` and `Opt.isNone` . It is kind of nice in this case, but this feature is best used on very long module names. Cases like this:

```
import Facebook.News.Story as Story
```

It would be annoying to type out the whole module name every time we need a function from it, so we shorten it to a name that is clear and helpful.

Exposing

You can also use the `exposing` keyword to bring in the contents of the module *without* a qualifier. You will sometimes see things like this:

```
import Optional exposing (Optional)

noService : Optional a -> Optional a -> Bool
noService shoes shirt =
  Optional.isNone shoes && Optional.isNone shirt
```

This way you can refer to the `Optional` type directly, but still need to say `Optional.isNone` and `Optional.None` for everything else exposed by the module.

This `exposing` keyword works just like it does in module declarations. If you want to expose everything you use `exposing (..)` . If you want to expose everything explicitly, you would say `exposing ( Optional(..), isNone )` .

Mixing Both

It is possible to use `as` and `exposing` together. You could write:

```
import Optional as Opt exposing (Optional)

noService : Optional a -> Optional a -> Bool
noService shoes shirt =
  Opt.isNone shoes && Opt.isNone shirt
```

No matter how you choose to `import` a module, you will only be able to refer to types and values that the module has made publicly available. You may get to see only one function from a module that has twenty. That is up to the author of the module!

Building Projects with Multiple Modules

We know what the Elm code looks like now, but how do we get `elm-make` to recognize our modules?

Every Elm project has an `elm-package.json` file at its root. It will look something like this:

```
{
    "version": "1.0.0",
    "summary": "helpful summary of your project, less than 80 characters",
    "repository": "https://github.com/user/project.git",
    "license": "BSD3",
    "source-directories": [
        "src",
        "benchmarks/src"
    ],
    "exposed-modules": [],
    "dependencies": {
        "elm-lang/core": "4.0.2 <= v < 5.0.0",
        "elm-lang/html": "1.1.0 <= v < 2.0.0"
    },
    "elm-version": "0.17.0 <= v < 0.18.0"
}
```

There are two important parts for us:

- `"source-directories"` — This is a list of all the directories that `elm-make` will search through to find modules. Saying `import Optional` means `elm-make` will search for `src/Optional.elm` and `benchmarks/src/Optional.elm`. Notice that the name of the module needs to match the name of the file exactly.
- `"dependencies"` — This lists all the community packages you depend on. Saying `import Optional` means `elm-make` will also search the `elm-lang/core` and `elm-lang/html` packages for modules named `Optional`.

Typically, you will say `"source-directories": [ "src" ]` and have your project set up like this:

```
my-project/elm-package.json
my-project/src/Main.elm
my-project/src/Optional.elm
```

And when you want to compile your `Main.elm` file, you say:

```
cd my-project
elm-make src/Main.elm
```

With this setup, `elm-make` will know exactly where to find the `Optional` module.

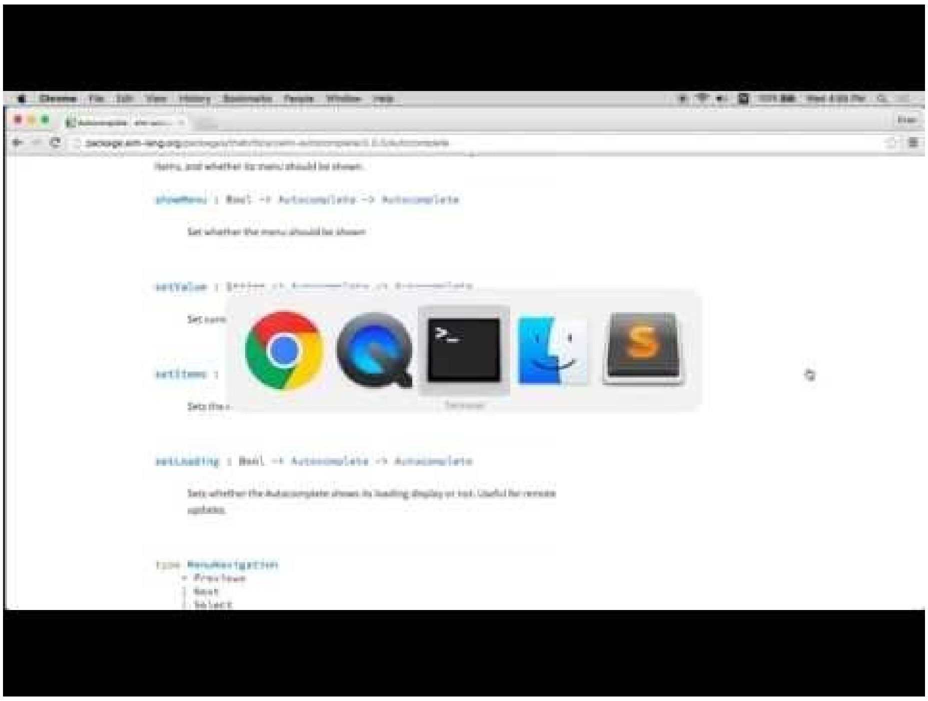
> **Note:** If you want fancier directory structure for your Elm code, you can use module names like `Facebook.Feed.Story`. That module would need to live in a file at `Facebook/Feed/Story.elm` so that the file name matches the module name.

More

Right now this section gives a brief introduction to **reusable views**. Instead of thinking about components, you create simple functions and configure them by passing in arguments. You can see the most extreme versions of this by checking out the following projects:

- `evancz/elm-sortable-table`
- `thebritican/elm-autocomplete`

The README of `elm-sortable-table` has some nice guidance on how to use APIs like these and why they are designed as they are. You can also watch the API design session where Greg and I figured out an API for `elm-autocomplete` here:



Video link

We talk through a lot of the design considerations that go into APIs like these. One big takeaway is that you should not expect to do something as elaborate as this for every single thing in your app! As of this writing, NoRedInk has more than 30k lines of Elm and one or two instances where they felt it made sense to design something as elaborate as this.

Hopefully those resources help guide you as you make larger and larger programs!

> **Note:** I plan to fill this section in with more examples of growing your `Model` and `update` functions. It is roughly the same ideas though. If you find yourself repeating yourself, maybe break things out. If a function gets too big, make a helper function. If you see related things, maybe move them to a module. But at the end of the day, it is not a huge deal if you let things get big. Elm is great at finding problems and making refactors easy, so it is not actually a huge deal if you have a bunch of entries in your `Model` because it does not seem better to break them out in some way. I will be writing more about this!